EuroOffice Extension Creator Documentation

Release 0.3

MultiRacio Ltd.

May 08, 2009

CONTENTS

Contents:

CHAPTER

ONE

DOWNLOADING AND INSTALLING

EuroOffice Extension Creator development is managed through Launchpad at https://launchpad.net/eoec/. Stable releases are available at https://launchpad.net/eoec/+download and the live source code can be checked out from the Bazaar repository `lp:eoec`.

Using Bazaar is surprisingly simple. Installation instructions and downloads for every platform are at http://doc.bazaar-vcs.org/latest/en/mini-tutorial/index.html#installation. Once it is installed, just execute

```
bzr branch lp:eoec
```

and the latest source code will be downloaded. Now you can edit this source code and make commits, but take note that these commits are only made to your local repository. When you have created something magnificent, you can upload it to Launchpad, and the EuroOffice Extension Creator maintainers can then take a look and accept the changes. This process is slightly more complicated than checking out, but it is well described at https://help.launchpad.net/Code/UploadingABranch.

Now that the source code is on your computer you can start using EuroOffice Extension Creator, so on to the next section!

CHAPTER

TWO

USING EUROOFFICE EXTENSION CREATOR

2.1 Creating a new extension

Use `create.py` to create your new OpenOffice.org extension. A command line such as the following can be used:

```
python create.py --vendor="My Name" "My Extension"
```

This will create a new directory named `My Extension` in the current directory, and fill it with the files necessary for a basic extension. This directory is now your development tree.

Note: OpenOffice.org does not take well to special characters in file names. Since the vendor name and extension name are both used as parts of directory names avoiding the use of special characters in either is recommended.

The created extension contains a number of default choices. For an explanation of the generated files see *The files in a new project*. While the recommended and most comfortable way of development is by customizing the code while OpenOffice.org is running, there are still a number of settings that can not be changed while the extension is loaded, and some that can only be changed prior to installation.

`create.py` also has a `--prefix=` option that allows you to set a custom prefix for your internal names, such as `com.sun.star` for Sun. It is recommended to set it to something like `com.mycompany` or `org.myorganization` to avoid name clashes in some instances. It defaults to `org.openoffice`.

2.2 Packing a development version OXT

When you are happy with the basic set up of the extension, you can create a development version OXT. The command line for this is:

```
python pack.py -D "My Extension"
```

(Execute this command in the parent directory of the extension, the same place where `create.py` was used.)

This will create a file named `My_Extension_Debug.oxt`. This is a valid OpenOffice.org extension now, and once it is installed, the development process can begin.

Note that this extension is now specific to your computer in that it includes the path to your development tree and will load its source code from there. This way you can perform all the development within this tree, and there is no need to copy files from one place to another. However it can not be shared with other developers.

To install an extension in OpenOffice.org open the **Extension Manager** dialog (from the *Tools* menu), press the **Add** button and pick the newly created OXT file. (To make installation simpler `pack.py` has a `-d` option using which the location of the created OXT file can be set.) You will see that already a license was generated (GPL), and `description.xml` is set up with a number of defaults (such as an icon).

Restart OpenOffice.org after installing the extension. (Make sure that you also close the quickstarter. You know you have really restarted OpenOffice.org if the splash is displayed on startup.)

Note: The prebuilt OXT files included in the release archives are not development versions, so it is not possible to try run-time development using them.

2.3 Development

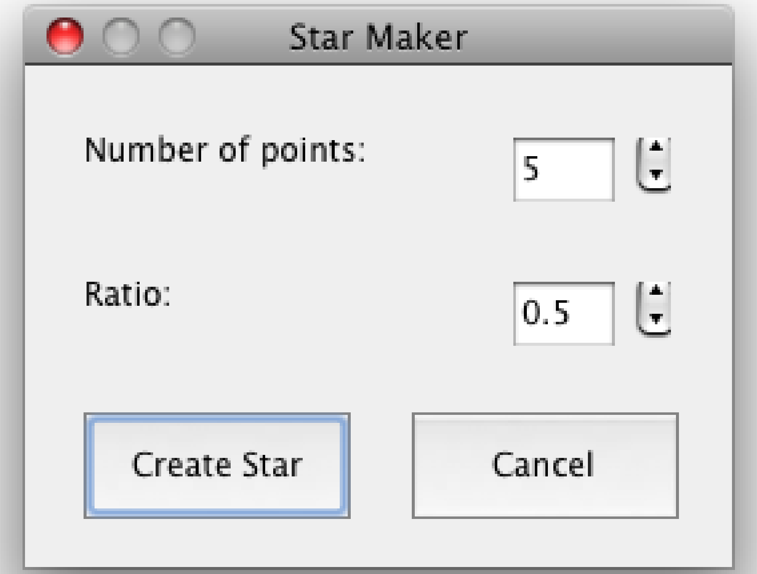
There are now two ways to introduce new code during run time.

To run a few arbitrary lines of code open the **EuroOffice Extension Creator** dialog. It can be accessed by pressing the Debug button on the **About** dialog of the extension, which in turn can be accessed from the Help menu. There is an empty textbox in the **EuroOffice Extension Creator** dialog where you can enter Python code to quickly execute it. It can be used for discovering the structure of OpenOffice.org objects, experimentation and for one-off code execution. Note that the contents of the textbox are not saved. The code is executed in the local context of the extension object and `self` is defined to be the object itself.

The other way is by editing the core source file of your extension, `myextension.py`. This file is monitored by EuroOffice Extension Creator for changes, and reloaded as needed. When the file is reloaded, even existing instances of the extension class are updated, so code changes will be instantly visible.

2.3.1 An example: Star Maker

In this example we want to create an extension that draws stars. The user can set the number of points the star has and the ratio between its inner and outer points (how “fat” the star is) through a dialog box.



Star Maker

Number of points: 5

Ratio: 0.5

We can start by designing the dialog box above.

For building dialogs we use the dialog designer in OpenOffice.org. Start it from the *Tools → Macros → Organize Dialogs* menu item. In the **OpenOffice.org Basic Macro Organizer** select `MyNameStarMakerDialogs` and click the **New** button. Give the name of the dialog – we will go with `NewStar` in the example. Then press Edit and the dialog editor will open with a very blank dialog.

The usage of the dialog editor is pretty self explanatory and not the topic of this tutorial, but a few things are useful to note:

- Make sure you give good internal names to the objects that you want to access programmatically. And since often it is not known in advance which objects are those it is best to give sensible names to everything.
- The tab order is important for a good user experience, so pay attention to it once the dialog looks good.
- You can select the window itself by clicking on its border (which is 1 pixel wide).
- Double click something to open the properties box.
- You can design dialogs made up of more than one page (for wizard style paged dialogs). For this select the window and set the page property to 1. Create page one. Set the page property of the window to 2. Everything on page one disappears and you can create page two. Note that every object has a page attribute, and they will only appear on that page. If the page attribute of an object is 0, it will appear on all pages.

  Make sure you set the page of the window to 1 once you are done.
- If you create **OK**, **Cancel** and **Help** buttons make sure you set their button type property accordingly. If you create a **Help** button, make sure you set the help URL property of the dialog window.

Click save once you are done in the dialog editor.

The EOEC dialog has a button labeled **Copy dialogs from the installed extension to your development branch**. It is a good idea to make a habit of pressing this button once the dialog is finished, because right now the design for the dialog only exists in OpenOffice.org and not in your development tree. Pressing this button copies the dialogs to the development tree.

Now we want a menu item in OpenOffice.org Draw that opens this dialog box. We want this menu item to be created when the user installs our extension, and we also want it to be created right now, so we do not have to reinstall our extension.

Open the `starmaker.py` file in your code editor. Take a look at the `firstrun` method. This method is executed on the first startup after you extension was installed. Currently it installs the *About Star Maker* menu item in the help menu.

The `addMenuItem()` method has four parameters:

1. The document type for which we want to install the menu item. For Draw it is `'com.sun.star.drawing.DrawingDocument'`.
2. The menu item after which we want to install our menu item, or if we want it to be the first in a menu, then the parent menu item. Menu items are referenced by their internal OpenOffice.org service names. You can find out the service name of every menu item by executing

   ```
   self.dumpMenus( 'com.sun.star.drawing.DrawingDocument' )
   ```

   in the EOEC dialog. We want to insert the new menu item at the top of the insert menu, so the service name is `'.uno:InsertMenu'`.
3. The name of the new menu item as it should appear. We can simply give a string here, or we could use `self.localize()` to get a localized label.

4. The internal name of the menu item, which is also the name of the method in the extension class that needs to be invoked when the menu item is clicked.

In the end the line to insert our new menu item is:

```
self.addMenuItem(
                'com.sun.star.drawing.DrawingDocument',
                '.uno:InsertMenu',
                'Star Maker',
                'starmaker' )
```

Try it in the EOEC dialog. If everything worked a new menu item should have appeared in the Insert menu (only in Draw). If it worked correctly, add this line to the `firstrun` method too, and to make sure we behave politely, add

```
self.removeMenuItem(
                'com.sun.star.drawing.DrawingDocument',
                'starmaker' )
```

in the `uninstall` method so that the menu item is removed when the extension is.

If you click the menu item now, nothing happens yet. We need to write the corresponding method, which is `starmaker`. Create a new method in the class named `starmaker` like this:

```
def starmaker( self ):
        self.box( 'Hello' )
```

Save `starmaker.py` and click the menu item again. The code will be reloaded and a message box displayed. We can now delete the placeholder `self.box()` call and start writing our code in earnest.

The first thing we want to do is display the dialog box that we meticulously designed:

```
dlg = self.createdialog( 'NewStar' )
ok = dlg.execute()
```

The `execute` method displays the dialog box and waits until it is closed. The dialog is closed when either the **OK** or the **Cancel** button is pressed or by some other means (using Esc or Alt-F4 on Windows for example). The return value of `execute` can be used to tell how the dialog was closed, and it is `True` when it was closed via the **OK** button.

If you save the source code and try the menu item now, you will see that the dialog you have designed pops up.

So we do not want to do anything if **OK** was not pressed:

```
if not ok:
        return
```

Otherwise we want to get the values entered into the two numeric fields (which we have named `PointsField` and `RatioField`):

```
points = int( dlg.getControl( 'PointsField' ).Value )
ratio = float( dlg.getControl( 'RatioField' ).Value )
```

You could now display these values in a message box to check if everything is working.

Now let us create an array of two-dimensional coordinates that make up our star:

```
poly = []
size = 1000
import math
for i in range( points ):
        # add an external point
        x = size * math.sin( i * math.pi * 2 / points )
        y = size * math.cos( i * math.pi * 2 / points )
        poly.append( (x, y) )
        # add an internal point
        x = ratio * size * math.sin( (i + 0.5) * math.pi * 2 / points )
        y = ratio * size * math.cos( (i + 0.5) * math.pi * 2 / points )
        poly.append( (x, y) )
```

Now we want to create a polygon object in Draw with these coordinates.

First we get a reference to the current document:

```
doc = self.getcomponent()
```

Then, since documents can be made up of multiple pages, we have to get the current page. Since this is not Impress, there is only one page, so it is page index 0:

```
page = doc.getDrawPages().getByIndex( 0 )
```

Now we create the polygon on this page using *utility library*:

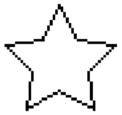
```
import util.draw
shape = util.draw.createPolygon( doc, page, [poly] )
```

A polygon in Draw can be made up of multiple outlines (this way they can have holes for example), so `createPolygon` expects a list of lists of coordinate pairs – this is why `poly` is put in a list here.

And because it is currently off the page, we move it to a sensible position:

```
setpos( shape, 10000, 10000 )
```

And we are done! Save the source code again and test!



2.3.2 About the About dialog

The source code of the **About** dialog is included in every newly created extension. This way you can customize it or remove it entirely, but it is also a useful example from which you can always copy parts into your own dialogs. A valuable example is how the **Debug** button works. The `addActionListener` method of the button is used to set up the listener:

```
dlg.DebugButton.addActionListener( self )
```

And then we create the method that will be called:

```
def on_action_DebugButton( self ):
```

Use this pattern to easily create button handlers. Such convenient name-based mechanisms are not available for every kind of event handler.

2.4 Packing a release version OXT

Before starting to pack a release version make sure that the dialogs that you have created or modified in the development version have been copied back to the development tree. The button labeled **Copy dialogs from the installed extension to your development branch** in the EOEC dialog can be used to perform this operation.

Once the development tree correctly represents the extension a release version can be packed by executing:

```
python pack.py "My Extension"
```

The created `My_Extension.oxt` file will be a release build ready for distribution.

2.5 Updating an extension

EuroOffice Extension Creator development is an ongoing process, and if you want to use features from a newer version you might want to be able to update your extension tree. This is possible with the `update.py` script.

To update simply execute:

```
python update.py "My Extension"
```

It works by comparing file modification dates, and if the template file is newer than the file in your development tree the file will be overwritten.

Warning: This command can destroy your work!
To make this less likely it first performs a dry-run and then asks for permission to really update the files. This cautiousness can be disabled with the `--quiet` option.

If a file is updated in both the EOEC templates and in your development tree, you may want to merge the changes. To do this rename the file in your tree (to `filename.mine` for example) and run `update.py`. A new file based on the template will be created and you can manually perform the merging.

To prevent a file from ever getting up list it in the `dontupdate` variable in the `eoec.config` file (see *The files in a new project*).

CHAPTER

THREE

THE EXTENSION CORE FACILITIES

The Extension Core is a set of functions that every EOEC extension depends on. It provides the backbone of creating and registering an extension, the run-time development mechanism, debugging and a number of essential utilities.

A large part of the functionality is present in the `ComponentBase` class, and some parts that are not tied to an extension object are implemented as functions.

3.1 The `ComponentBase` class

class `ComponentBase``()`

This is the base class from which extension classes are derived. `ComponentBase` itself is in turn derived from `unohelper.Base`, the base class provided by PyUNO.

The following interfaces are implemented:

- XComponent
- XInitialization
- XJob
- XJobExecutor
- XServiceDisplayName
- XServiceInfo
- XServiceName

The following methods can be overridden by subclasses to customize their behavior:

`startup``()`

Runs at every time OpenOffice.org starts up. Subclasses may override it to do some kind of initialization here – the default implementation does not do anything.

Try to write your extension so that most of the heavy initialization only happens when it is needed. This way slowing down startup unnecessarily can be avoided.

`firstrun``()`

Runs the first time OpenOffice.org starts up after the extension was installed. Subclasses may override it to do some kind of initialization here – the default implementation does not do anything.

This is the recommended place to create new menu items.

`uninstall``()`

Runs when the user presses the **Disable** or **Remove** buttons on the extension in the Extension Manager dialog. Subclasses may override it to do some kind of clean up here – the default implementation does not do anything.

This is the recommended place to remove anything that was created in `firstrun()`.

The following utility methods are provided:

getcontroller()

Returns the controller associated with the current document. For example if the current document is a `com.sun.star.TextDocument` `getcontroller()` will return the associated `com.sun.star.TextDocumentView`.

addMenuItem(*documenttype, menu, title, command, submenu=False, inside=True*)

Adds a new menu item to the OpenOffice.org menu system. This addition is permanent. The menu item can only be removed using `removeMenuItem()`. It is recommended to add menu items in the `firstrun()` method and remove all of them in the `uninstall()` method.

An example creating a new submenu with two menu items in the **Insert** menu:

```
self.addMenuItem('com.sun.star.text.TextDocument',
                '.uno:InsertMenu', self.localize('submenu'), 'mysubmenu', submenu=True)
self.addMenuItem('com.sun.star.text.TextDocument',
                'mysubmenu', self.localize('item-2'), 'method2')
self.addMenuItem('com.sun.star.text.TextDocument',
                'mysubmenu', self.localize('item-1'), 'method1')
```

The menu items are added in reverse order, so `item-1` will be the top item and `item-2` the bottom.

Icons can be assigned to menu items through the xml configuration file `Addons.xcu`.

Parameters
- *documenttype* (string) – a document type, such as `'com.sun.star.text.TextDocument'` (see `documenttypes`)
- *menu* (string) – the command URL of the menu item under which (or into which) the new menu item is to be placed
- *title* (string) – The label to appear to the user. It is recommended to use `localize()` to get a localized menu item.
- *command* (string) – The internal name of the menu item. For normal menu items this is also the name of the method to be called when the menu item is clicked. For submenus this is the command URL that can be used as the `menu` parameter when adding menu items into it. If `command` contains a colon (:) it is treated as a command URL.
- *submenu* (boolean) – If true a new submenu, that can contain menu items, is created.
- *inside* (boolean) – If true and `menu` refers to a submenu, the new item is created inside this submenu. Otherwise the new item is created after the item referred to by `menu` even if it is a submenu.

removeMenuItem(*documenttype, command, submenu=False*)

Removes the menu item or submenu from the menu tree of the given document type.

Parameters
- *documenttype* (string) – a document type, such as `'com.sun.star.text.TextDocument'` (see `documenttypes`)
- *command* (string) – the internal name of the menu item
- *submenu* (boolean) – set to true when removing submenus

box(*message, kind = 'infobox', buttons = 'OK', title = None*)

Display a message box. Uses `com.sun.star.awt.Toolkit.createMessageBox()`, but saves a lot of the verbosity of the UNO API. The constants used come from the UNO API and are not validated by this method.

The `infobox` can only display an **OK** button, so if `buttons` is not `'OK'`, kind defaults to `'querybox'` instead.

Parameters
- *message* (string) – the text to display
- *kind* – one of `'infobox'`, `'warningbox'`, `'errorbox'`, `'querybox'` and `'messbox'`
- *buttons* – one of `'OK'`, `'OK_CANCEL'`, `'YES_NO'`, `'YES_NO_CANCEL'`, `'RETRY_CANCEL'` and `'ABORT_IGNORE_RETRY'`

- *title* – The title of the message box. Defaults to the localized extension name if `None`.

Return type integer

Returns The result of the `execute()` call on the message box. Different values are returned depending on which button was pressed. The `BOXCANCEL, BOXOK, BOXYES, BOXNO, BOXRETRY` attributes can be used to distinguish. `BOXCANCEL` has a `False` truth value, so simple `OK_CANCEL` boxes can be evaluated without the use of these constants.

`debugexception_and_box`(*format = None*)

This method can be used in place of `debugexception()` where even in release mode the user should be alerted of the problem.

`format` can be used to customize the error message. By default it is

```
'An unexpected error (%(kind)s) occured at line %(linenumber)s of %(filename)s.'
```

The format string can contain insertion points for `kind` (the exception type), `filename`, `linenumber`, `functionname` and `text` (the message of the exception).

This method also calls `debugexception()` so in debug mode the entire stack trace is recorded.

It is recommended to add instructions for the user in the format string (such as “Please let us know about this problem at support@example.com.”).

`dumpMenus`(*documenttype*)

This method outputs a representation of the complete menu system of the given document type to the debug output. It is useful for finding the right command URL when placing your menu items.

`getconfig`(*nodepath, update=False*)

Returns the `com.sun.star.configuration.ConfigurationAccess` object associated with the given node path.

See the `config` attribute.

Parameter *update* – If `True`, a `ConfigurationUpdateAccess` object is instead returned, that can be used to change configuration settings.

`localize`(*string, language=None*)

Returns the localized string associated with the internal string `string`.

If `language` is `None`, the language is automatically chosen based on the `SUPPORTED_LANGUAGES` attribute. If the current user interface language (as given by `org.openoffice.Setup.L10N.ooLocale`) is in the list of supported languages, it is used, otherwise the first element of the list (the fallback language) is used in its place.

If `language` is not `None`, it should be the two-letter code of a language to be used instead of the default.

The localizations are loaded from `MyNameMyExtensionDialogs/DialogStrings_language.properties` (with the two-letter code for the language). See *the description of this file* .

Returns Normally the localized string is returned. However if there is no localization for the given string in any language, the string `'unlocalized: '+string` is returned and if there are localizations just not in the chosen language, the string `'unlocalized for %s: %s'%(language, string)` is returned instead. These can be used to quickly spot missing translations on the user interface.

`createdialog`(*dialogname*)

Creates an instance of the given dialog using `com.sun.star.awt.DialogProvider.createDialog()`.

For increased convenience, the dialog controls are accessible as attributes of the returned object. For example if the dialog `Dialog1` has a button named `Button1` adding an ActionListener can be done like this:

```
dlg = self.createdialog( 'Dialog1' )
dlg.Button1.addActionListener( self )
```

This is mostly equivalent to `dlg.getControl( 'Button1' ).addActionListener( self )`, but is more efficient when the control is accessed multiple times, because each time an UNO call is saved.

For technical reasons a wrapper object is created around the `com.sun.star.awt.XDialog` object to enable this convenient access. The original `XDialog` object can be accessed as the `xdialog` attribute of the wrapper. The wrapper forwards every attribute access to the `XDialog` object, so accessing the original object is practically never necessary.

To display a dialog, call its `execute()` method.

Parameter *dialogname* – the name of the dialog that you have used in the OpenOffice.org dialog designer

Returns an instance of the given dialog

`getdesktop``()`

Returns the `com.sun.star.frame.Desktop` object.

`getcomponent``()`

Returns the currently active document object as returned by `com.sun.star.frame.Desktop.getCurrentCompone` When there is no current component for some reason, an arbitrary open document is returned. If there are no open documents, `None` is returned.

`getcontroller``()`

Returns the view associated with the currently active document object as returned by `com.sun.star.frame.Desktop.getCurrentComponent().getCurrentController()`.

The `ComponentBase` class also sets up a few attributes that can be useful:

`ctx`

The `com.sun.star.uno.XComponentContext` object that was given to the constructor. It can be used to get the "Service Manager" object which can instantiate a number of object types.

`path`

Contains the root path of the installed extension. It is a normal path, not a file URL.

`uilanguage`

The two-letter code of the UI language of OpenOffice.org. If the actual UI language is not in the `SUPPORTED_LANGUAGES` list, then `uilanguage` is instead set to the fallback language (the first language in the list).

`SUPPORTED_LANGUAGES`

This attribute is set by subclasses to a list of the two-letter language codes of supported languages. For languages not in this list, the fallback language (the first language in the list) is used.

`config`

Provides access to the persistent configuration of the extension. It is a `com.sun.star.configuration.ConfigurationUpdateAccess` object associated with the node `my.prefix.MyExtensionSettings/ConfigNode`.

Keep in mind that for the change to the configuration to be persistent, the `commitChanges()` method has to be called.

For example on the first start up (after `firstrun()` was called) the `FirstRun` configuration setting is updated like this:

```
self.config.FirstRun = False
self.config.commitChanges()
```

`BOXCANCEL, BOXOK, BOXYES, BOXNO, BOXRETRY`

These are the possible return values from the `box()` method. `BOXCANCEL` is the only one of these with a `False` truth value.

3.2 The debug framework

The Extension Core provides a basic debugging framework. Since debugging is most important when things go wrong, and when things go wrong there is not much that can be relied on, debug messages are output to either `stderr` (on Linux) or to a debug file (`c:\debug.txt` on Windows and `/tmp/debug.txt` on Mac).

To provide some degree of convenience, the EOEC dialog displays the contents of the debug file (if there is one).

On platforms with debug files the debug file is truncated on startup and then on calls to the `debug()` function it is opened, appended and closed.

Note that all debug output is silently omitted if the extension is not in development mode.

The relevant functions are:

`debug`(**msgs*)

The list of objects in `msg` is written to the debug output on separate lines. The objects are first converted to unicode strings and then encoded in UTF-8.

`dd`(**args*)

For each of the objects in `args` writes to the debug output the result of `dir(obj)`. It is a shorthand for discovering what methods and attributes a UNO object has.

`debugexception`()

Writes the last exception (with stack trace) to the debug output.

You can notice in the source code of the Extension Core and the blank extension created by `create.py` that calls from the "outside" are wrapped in try-except-debugexception blocks. This is very useful for debugging, because otherwise the exception is propagated to OpenOffice.org and will most likely result in a crash that gives little information. It is recommended that in new methods that implement UNO interfaces extension developers follow the same convention even when the method does not appear to do anything "dangerous".

An example of this important principle:

```
# XActionListener
def actionPerformed(self, event):
        try:

                pass            # suppose we do something...

        except:
                debugexception()
```

`debugstack`()

Writes the current stack trace to the debug output.

3.3 Utilities

The Extension Core also includes a number of useful functions and constants that are used by `ComponentBase` and can also be useful to developers of extensions.

Normally extensions import * from `extensioncore`, so these utilities are conveniently available.

`DEBUG`

This boolean tells if we are in development mode.

class **`SelfUpdating`**()

This base-class provides a modified `__getattribute__` implementation that (when in development mode) checks if a newer version of the source file is available (based on file modification dates) and reloads it if necessary, modifying this instance and future instances of the class.

`ComponentBase` is derived from this class, but if other classes need this functionality they can derive from `SelfUpdating` too.

`props`(**args, **kwargs*)

The UNO API often expects parameters to be `com.sun.star.beans.PropertyValue` arrays. This function makes it easy to create such arrays.

`props` can be called with either a single dictionary object as the parameter, or with keyword arguments. It will then turn the given dictionary or the keyword arguments into a `com.sun.star.beans.PropertyValue` array and return this array.

`anyprops`(**args, **kwargs*)

The same as `props()` except the resulting array is wrapped into an `Any` object, which is needed in some UNO calls.

`namedvalues`(**args, **kwargs*)

Same as `props()` except with `NamedValue` arrays instead of `PropertyValue` arrays.

`anynamedvalues`(**args, **kwargs*)

Same as `anyprops()` except with `NamedValue` arrays instead of `PropertyValue` arrays.

`unprops`(*props*)

The UNO API often returns values as `com.sun.star.beans.PropertyValue` arrays. `unprops` unpacks these arrays into a more convenient format.

The class of the returned object is derived from `dict`, so the results are accessible as in a normal dictionary object. However for added convenience it is also possible to access the values as attributes of the returned object.

For example to access the command URL of the first menu item in some menu system:

```
settings = xUIMgr.getSettings('private:resource/menubar/menubar', True)
menu = unprops(settings.getByIndex(0))
debug(menu.CommandURL)
```

This function also works with `NamedValue` arrays.

`enumerate`(*obj*)

UNO `Sequence` objects are converted to tuples when they come to the Python side and are easy to work with. However the OpenOffice.org API has a number of different container types that are not automatically converted and can be less convenient to work with.

`enumerate` is a generator function for these container types. An example usage for relabeling footnotes:

```
for footnote in enumerate( self.getcomponent().Footnotes ):
        footnote.Label = 'Footnote ' + footnote.Label
```

The following kinds of objects are supported:

- `com.sun.star.container.XEnumerationAccess`
- `com.sun.star.container.XEnumeration`
- `com.sun.star.container.XIndexAccess`
- `com.sun.star.container.XNameAccess` (while this is a dictionary-like container only the values are yielded not the keys)

`safeeval`(*code*)

Evaluates a Python expression making sure it is safe.

The expression can contain only a very limited set of language elements, that are only sufficient for describing objects made up of basic Python types. This way a dictionary mapping strings to lists of integers for example can be simply stored by calling `repr()` on it, and can be loaded using this function.

`documenttypes`

This variable lists the possible document types (such as `'com.sun.star.text.TextDocument'`) in OpenOffice.org.

Note that not every build of OpenOffice.org has all of these document types!

class **`propset`**`()`

This class is a lightweight implementation of `com.sun.star.beans.XPropertySet` and `com.sun.star.lang.XServiceInfo`. None of the methods in the `XPropertySet` interface are actually implemented except for `getPropertyValue()` and `setPropertyValue()`. This is appropriate for most purposes, such as for inserting a new item into a `com.sun.star.ui.ActionTriggerContainer`. Error handling is also not implemented.

`__init__``(`**args, **kwargs*`)`

The constructor can be called with either a single dictionary object as the parameter, or with keyword arguments. It will then use the given dictionary or the keyword arguments as the initial property set.

`init``(`*cls, *services*`)`

Registers the given class in the OpenOffice.org type database for the given services. This function should not be called arbitrarily – it is a part of the extension initialization process. It is called in the last line of `extensionname.py`. Extensions automatically support the `com.sun.star.task.Job` service but if further services are needed to be registered for they can be added to this function call.

Of course the required interfaces should be implemented when registering for a service.

Parameters
- *cls* (class) – the class to register
- *services* (strings) – the names of the services the class supports

CHAPTER

FOUR

THE UTILITY LIBRARY

EOEC speeds up development, but given how complicated the OpenOffice.org API is in some parts it is also very important to facilitate code reuse, and this is why we have started to collect the most often used pieces of code in a supplemental API package. This is the `util` module that you can access from your extensions.

4.1 `util.draw`

The functions in this module deal with UNO API related to Draw objects.

The unit of measurement used for coordinates in the UNO API is one thousandth of an inch. It is also used internally in OpenOffice.org and only integer coordinates are allowed.

`rgb`(*r, g, b*)

Converts a color given in float representation to the format used in the UNO API. For example `rgb(1.0, 0.0, 0.0)` is red.

Parameters
- *r* (float) – red component in range 0.0 - 1.0
- *g* (float) – green component in range 0.0 - 1.0
- *b* (float) – blue component in range 0.0 - 1.0

Return type int

Returns the integer representation of the color suitable for use with the UNO API

`RGB`(*r, g, b*)

Converts a color given in integer representation to the format used in the UNO API. For example `rgb(255, 0, 0)` is red.

Parameters
- *r* (int) – red component in range 0 - 255
- *g* (int) – green component in range 0 - 255
- *b* (int) – blue component in range 0 - 255

Return type int

Returns the integer representation of the color suitable for use with the UNO API

`setpos`(*shape, x, y, w=None, h=None*)

Moves and optionally resizes a Draw object.

Parameters
- *shape* – the shape to move
- *x, y* (int) – destination coordinates

`createShape`(*model, page, shapetype, color=None*)

Creates a new shape and adds it to the DrawPage.

The shape is created by:

```
model.createInstance( 'com.sun.star.drawing.%sShape'%shapetype )
```

Parameters
- *model* (`com.sun.star.lang.XMultiServiceFactory`) – The document object that can be used to create an instance of the given shape type. Normally it is the object you got the DrawPage from.
- *page* (`com.sun.star.draw.DrawPage`) – The page to insert the new shape into. Normally one of the DrawPages of `model`.
- *shapetype* (string) – the kind of shape to be created
- *color* (int or `None`) – the fill color or `None` for a transparent shape

`createPolygon`(*model, page, coordss, color=None, type='PolyPolygon'*)

Uses `createShape` to create a `com.sun.star.drawing.PolyPolygonShape` or `com.sun.star.drawing.PolyLineShape`. The "PolyPolygon" shape is made up of the combination of a number of polygons, each of which are a closed outline. (The polygons are implicitly closed, there is no need to duplicate the first point.) The "PolyLine" shape is a collection of line sequences.

Parameters
- *model* (`com.sun.star.lang.XMultiServiceFactory`) – The document object that can be used to create an instance of the given shape type. Normally it is the object you got the DrawPage from.
- *page* (`com.sun.star.draw.DrawPage`) – The page to insert the new shape into. Normally one of the DrawPages of `model`.
- *coordss* (list of lists of pairs of integers) – The list of polygons each described by a list of their points which in turn are described by their integer coordinates. For example `[[(0, 0), (1, 0), (1, 1), (0, 1)]]` is a unit sized square.
- *color* (int or `None`) – the fill color or `None` for a transparent shape
- *type* (string) – `'PolyPolygon'` (the default, for drawing closed polygons) or `'PolyLine'` (for drawing open line segments).

`embed`(*doc, imagefilename*)

Inserts the image into the document as an embedded image. As opposed to a linked image it will be saved into the document instead of becoming an external dependency.

Parameters
- *doc* (`com.sun.star.lang.XMultiServiceFactory`) – The document object to insert into, for example a `com.sun.star.text.TextDocument`.
- *imagefilename* (string) – The name of the image file in system-specific format (not as a URL).

4.2 `util.writer`

The functions in this module deal with UNO API related to Writer documents.

`getWord`(*view*)

Gets the word under the text cursor. Returns a cursor object for the word which you can then use to access and modify the word:

```
import util.writer
cursor = util.writer.getWord(self.getcontroller())
if cursor.String == 'something':
        # replace something with something else
        cursor.String = 'something else'
```

Parameter *view* (`com.sun.star.text.XTextViewCursorSupplier`) – The current `TextDocumentView` typically as returned by `extensioncore.ComponentBase.getcontroller()`.

Return type `com.sun.star.text.XTextViewCursor`

Returns A cursor object for the word.

4.3 `util.web`

Since the web contains such a large number of useful sources of information and services, an extension might want to make efficient use of it. The goal of this module is to help with this.

`loadpage`(*url*)

Retrieves the page with the given URL.

`urllib2` is used and the `User-agent` HTTP header is set to:

```
Mozilla/5.0 (OpenOffice.org extension "My Extension" created with EuroOffice Extension Creator)
```

The extension name is injected by the `create.py` script. To use a different `User-agent` header, `util/web.py` will have to be modified.

Parameter *url* (string) – URL of the page to retrieve

Return type string

Returns contents of the web page

`getpage`(*url*)

Retrieves the page with the given URL.

This function uses a global cache object. If the given URL has already been retrieved by `getpage()`, it will be served from a persistent cache. If it has not yet been cached, it is retrieved with `loadpage()` and added to the cache.

Parameter *url* (string) – URL of the page to retrieve

Return type string

Returns contents of the web page

`flatten`(*x*)

Flatten a Beautiful Soup object. Sometimes nothing but the bare text content is wanted from part of a web page, and this function supplements the Beautiful Soup API to provide access to it.

Parameter *x* – the Beautiful Soup object

Return type string

Returns the flattened string

`MAX_CACHE_SIZE`

The maximal cache size. Once the cache reaches this size it is deleted and a new cache is started.

Defaults to 10MB.

`setcachefile`(*filename*)

The file name to use for the web cache. By default the file name `'./cache'` is used, but this can depend highly on platform and OpenOffice.org build, so setting the cache file name is important.

If the file named does not exist a new file will be created. Take care, because if the file exists, but is not actually a cache file it will be overwritten with a cache file.

CHAPTER

FIVE

THE FILES IN A NEW PROJECT

`create.py` creates a number of files for your project. This section describes them in detail.

Since some of the directory names depend on your vendor and extension names, we will assume that the vendor name is `My Name` and the extension name is `My Extension`.

myextension

This is the most important directory for the development, because it contains the source code of the extension. The files here are:

myextension.py

This is the most important file for the developer, because it contains the customized source code of the extension. For simple extensions you will only have to edit this file. Also note that only this file is used for run-time development, so while you can have external code in other `.py` files in this directory, they will not be reloaded on changes.

extensioncore.py

This file includes all the core technology for your extension, most importantly its base class `ComponentBase`, the `init` function that takes care of registering the derived class with OpenOffice.org and the debugging framework.

See *The Extension Core facilities*.

__init__.py

This file is just here for technical reasons.

generatedsettings.py

This file is created by the `pack.py` script. It tells the extension whether it is in development or release mode, and in development mode also contains the path to the development source tree.

It is recommended not to put this file under version control.

util

This directory includes an optional, but hopefully useful library of utility modules. See *The Utility Library*.

The Utility Library is quite small at the moment, but once it expands, you may wish to remove it or trim it to only include the parts needed by your extension. This is possible, because the `extensioncore` module does not depend on it.

binaries-darwin-python23, binaries-linux-python23, binaries-windows-python23

These directories are included in the module search path when the OpenOffice.org installation uses Python 2.3. Python 2.3 does not include `ctypes`, so that is contained in this directory. You can also add here any other binary modules that your extension needs.

Similar directories can also be created for Python 2.5. It makes sense when you want to include a module like PIL with your extension or if you have created your own Python extension (though using `ctypes` is often a simpler option).

modules-python23

This directory contains platform independent Python modules that depend on the version of Python. By default the `compiler` and `ctypes` modules are placed here, because they are not included in the Python 2.3 in OpenOffice.org.

If a similar directory is created for Python 2.5 it will be appropriately handled.

`myextension-loader.py`

This is the file that is actually registered with OpenOffice.org. The code inside is responsible for setting up the appropriate module paths and setting up the connection between OpenOffice.org and the actual extension class in `myextension.py`.

Normally extension developers do not need to deal with this file, but there may be some special cases. If something goes wrong during the installation of the extension, the `DEBUG` option may be turned on in this file to learn more about the error. If additional Python versions (besides 2.3 and 2.5) are to be supported, this file also needs to be modified.

`MyExtension.xcs`

This file defines the configuration options used by the extension. It already contains two options, `Origin` which is used to find out the path of the extension and `FirstRun` which is used to execute installation code only once. More options can be easily added.

`MyExtension.xcu`

Similar to `MyExtension.xcs` this file contains the initial values for the configuration options. When adding new options make sure it is added in both files.

`Addons.xcu`

This file contains the icon associations for the new menu items in a rather verbose XML format. An icon (`menuicon_small.png` + `menuicon_large.png`) is associated with the *About* menu item and new associations can be created based on it.

`Jobs.xcu`

This file is used to register the extension to run at startup. If there is no need for this, it may make sense to skip it altogether, so that the extension does not impact the application start up speed. Removing the entry for `Jobs.xcu` in the `manifest.xml` is the simplest way of doing that.

`description.xml`

Contains important information about the extension and is recommended to be customized. It is filled with default values and placeholders, so customization is more or less straightforward, but a detailed documentation of the elements used can be found in the OpenOffice.org wiki.

`MyNameMyExtensionDialogs`

This directory contains the dialog library of the extension.

- **`About.xdl`**

  The **About** dialog.

- **`dialog.xlb`**

  The XML file describing the dialog library.

- **`DialogStrings_en_US.default`**

  This file marks English as the default language of the dialog library.

- **`DialogStrings_en_US.properties`**

  The localization file for the English language. It is a Java property resource bundle file, so tools like The Translate Toolkit can be used to handle it. (It can also be edited manually, but this is error prone.)

  This file contains the whole of the localization for the extension – not just the part concerning the dialogs. The `localize()` method also works with this file.

- **`ExtensionCreator.xdl`**

  The **EuroOffice Extension Creator** dialog.

`META-INF`

manifest.xml
When an extension is installed, every file in the archive is copied to the extension directory. In addition `manifest.xml` tells OpenOffice.org which of these files need special attention.

Unless new `.xcu` configuration files are used (such as when a new toolbar is defined) or new UNO types are registered, this file probably does not need to be modified. New Python modules, images, dialogs, help pages, etc. do not need to be added here.

help
This directory contains the help pages for the extension.

resources
The license files are placed here.

eoec.config
This Python source file is used as a configuration file that stores the information that was used to create the extension with `create.py` (such as the URL of the extension website and the name of the extension vendor). It is used by `update.py`.

To prevent a file from ever getting updated a `dontupdate` variable can be added to `eoec.config`. This variable should be a list of file names to skip during an update. (File names without paths should be listed. The Barcode example demonstrates this feature.)

A number of default image files are included. Replacing them is an easy way of customizing an extension.

extensionicon.png
The icon displayed in the **Extension Manager** dialog.

logo_en.gif
The image displayed in the **About** dialog for the English localization. Similar image files can be added for every supported language.

menuicon_large.png, menuicon_small.png
A pair of menu icons registered in `Addons.xcu` for the *About* menu item.

CHAPTER

SIX

EXAMPLE EXTENSIONS

6.1 Star Maker

Star Maker is a basic OpenOffice.org Draw extension that draws a star based on input from a dialog.

This is a good starter example and it is thoroughly documented in *An example: Star Maker*. There is also a screencast that uses this example to demonstrate EOEC development.

6.2 Barcode

This example is basically the same as Star Maker, but instead of stars it draws barcodes. Since there is a much larger demand for barcodes, this extension has become practically useful and with the help of many users it has been translated to numerous languages and many barcode types have been added.

It is now a good example of a full-fledged extension created with EOEC.

6.3 Merriam-Webster Dictionary

Merriam-Webster is a trademark of Merriam-Webster Incorporated. This example is just for developers and should not be released separately.

Merriam-Webster Dictionary is an OpenOffice.org Writer extension that integrates an online thesaurus. It is an example of an interactive dialog where words are looked up based on selection from a listbox and online connectivity. Python extensions in OpenOffice.org have the full power of Python available to them, so web pages can be fetched and processed with ease. For processing web pages this example uses Beautiful Soup, a very convenient and reliable pure-Python HTML parser.

6.4 Lookup

Lookup is a very simple extension that looks up the selected text (in OpenOffice.org Writer) on Google in the user's default browser. To make it a bit more interesting it illustrates how a keyboard shortcut can be assigned to a menu item.

6.5 Shuffle

Shuffle is an OpenOffice.org Calc example and illustrates basic cell access. It shuffles the rows of the current selection.

6.6 Sharpen

Sharpen is an OpenOffice.org Draw extension. It illustrates context-menu integration, the integration of a binary module (PIL) in a multiplatform way and access to Draw objects. It uses `util.draw.embed()` to insert the sharpened image back to OpenOffice.org.

CHAPTER
SEVEN

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

INDEX