

FLExTools Programming Help

For FLExTools Version 2.0,
and FieldWorks version 9.

May 2022

Introduction.....	2
Creating a Module.....	2
Reporting Functions.....	3
Messages.....	3
Progress Indication.....	3
Fieldworks Database Functions.....	4
Basic Database Functions.....	4
Advanced Lexicon Access.....	5
Writing Systems.....	5
Text Fields.....	6
Custom Fields.....	6
Texts.....	7
Going Deeper.....	7
Modifying the Fieldworks Database.....	8
Changing Strings.....	9
Flag Field.....	9
Documenting the Module.....	10
Debugging.....	11
Run Stand-alone.....	11
Extra Debugging Output.....	11

Introduction

FLExTools is a framework for running Python scripts on a FieldWorks Language Explorer database. FLExTools provides core functions for creating Python scripts (*Modules*) to do the processing work, and a user interface to make it easy to run the Modules individually or as a set.

This guide explains how to write your own FLExTools Modules. It assumes you are familiar with using FLExTools as described in *FLExTools Help.pdf* (accessed from the menu Help | Help.) Extra help on useful functions provided with FLExTools can be found on the menu Help | API Help.

Creating a Module

The following code fragment shows a minimal module with all the required pieces in place.

```
from FTModuleClass import *

docs = {FTM_Name      : "Demo",
        FTM_Version   : 1,
        FTM_ModifiesDB : False,
        FTM_Synopsis    : "Does nothing",
        FTM_Help       : None,
        FTM_Description: ""

def Demo(DB, report, modifyAllowed=False):
    report.Info("This module does nothing!")

FlexToolsModule = FlexToolsModuleClass(runFunction = Demo,
                                       docs = docs)
```

A FlexToolsModuleClass is initialised with the main processing function and the documentation. The instance must be called FlexToolsModule.

Notice the three parameters to the *Demo* function:

- DB—this is a FLExProject object, giving access to the LCM¹ Cache and helper functions for reading and writing the database.
- report—message reporting functions and progress indication.

¹Language and Culture Model; formerly called Fieldworks Data Objects (FDO)




- `modifyAllowed`—a flag to tell the Module whether the user wants to do a dry-run (`modifyAllowed==False`), or make changes to the database (`modifyAllowed==True`). If `modifyAllowed` is `True`, then FLEXTools will make a call to `undo stack` before running the Module.

These parameters are discussed in more detail in the following sections.

Reporting Functions

Messages

Message functions are available for reporting progress, warnings and errors. These messages are displayed in the bottom message pane of the FLEXTools main window with a different icon for each one. The functions are available through the second parameter to the processing function as follows:

 <code>report.Info(msg, info)</code>	A basic information message.
 <code>report.Warning(msg, info)</code>	A non-critical warning such as telling the user about data changes or integrity issues.
 <code>report.Error(msg, info)</code>	A critical situation that prevents further processing.
<code>report.Blank()</code>	A blank line with no icon and no text.

The `info` parameter is an optional string giving more details. This is provided in a tool tip when the user hovers over the message. If the message is referring to a particular lexical entry or wordform in the Fieldworks database you can pass `DB.BuildGotoURL(entry)` as the `info` parameter. The user can then double-click on the message and Fieldworks will open up at that entry.

Progress Indication

Use the following functions to display a progress indicator in the status bar:

<code>report.ProgressStart(max)</code>	Start the progress indicator, setting the maximum value to <i>max</i> .
<code>report.ProgressUpdate(value)</code>	Update the progress indicator to <i>value</i> . (<i>value</i> = [0...max-1])

For example:

```
report.ProgressStart(DB.LexiconNumberOfEntries())
for entryNumber, entry in enumerate(DB.LexiconAllEntries()):
    report.ProgressUpdate(entryNumber)
    #Do something with entry
```

Fieldworks Database Functions

The Fieldworks database is accessed via the DB parameter. DB is an instance of the FLEXPProject class. The class definition documentation can be found on the menu Help | API Help. This class contains many helper functions for accessing the Fieldworks database via the FieldWorks LCM interface.

Basic Database Functions

```
for e in DB.LexiconAllEntries():
    for sense in e.SensesOS:
        if not DB.LexiconGetSenseDefinition(sense):
            report.Warning("Missing definition in %s"
                           % DB.LexiconGetLexemeForm(e),
                           DB.BuildGotoURL(e))
        if len(sense.ExamplesOS) == 0:
            report.Warning("No examples in %s"
                           % DB.LexiconGetLexemeForm(e),
                           DB.BuildGotoURL(e))
```

This example shows the use of LexiconAllEntries() to iterate over the whole lexicon, plus how to traverse the senses within each entry. Notice the warning messages, which use LexiconGetLexemeForm() to tell the user which entry has the problem. BuildGotoURL() is also used to allow the user to double-click on the message to jump to the right location in Fieldworks.

For testing your algorithm, a subset of the lexicon can be traversed using a Python slice like this (looking at the first twenty entries):

```
for e in DB.LexiconAllEntries()[:20]:
```

Gloss, part of speech ('grammatical category' in Fieldworks terminology), and semantic domains can be retrieved with these functions:

```
DB.LexiconGetSenseGloss(sense, languageTagOrHandle=None)
DB.LexiconGetSensePOS(sense)
DB.LexiconGetSenseSemanticDomains(sense)
```

Examples are a list and accessed like this (assuming context from above):

```
for example in sense.ExamplesOS:
    Report.Info("Example %s" % DB.LexiconGetExample(example))
```

Note that LexiconGetSenseGloss(), LexiconGetLexemeForm(), LexiconGetSenseDefinition(), and LexiconGetExample() all use the default analysis or vernacular writing system as appropriate. These functions can take a second parameter to specify a different writing system.

Advanced Lexicon Operations

The above example traverses the lexicon in an un-specified order. If you need random access or wish to do more complex things it may be useful to load the lexicon into a Python dictionary. The following example illustrates how to do this. Populate *data* with the relevant fields that you need for your processing.

```
LexiconEntries = {}
for e in DB.LexiconAllEntries():
    # Add fields of interest to 'data'
    data = {}
    data['hvo'] = e.Hvo
    glosses = []
    for sense in e.SensesOS :
        glosses.append(DB.LexiconGetSenseGloss(sense))
    data['glosses'] = glosses
    # The key is a 2-tuple: (lexeme-form, homograph-number)
    LexiconEntries[(DB.LexiconGetLexemeForm(e),
                    e.HomographNumber)] = data
```

To traverse this dictionary sorted by lexeme form you can use:

```
for e in sorted(LexiconEntries.items()):
    print e[0], e[1]      # lexeme-form, homograph-number
```

Writing Systems

The following code fragment reports all the vernacular writing systems in use in the database, showing the display name, the language tag (e.g. 'en'), and writing system handle.

```
report.Info("Vernacular Writing Systems:")
for tag in DB.GetAllVernacularWSs():
    report.Info(u"    %s [%s, %i]" %
               (DB.WSUIName(tag), tag, DB.WSHandle(tag)))
```

GetAllAnalysisWSs() is also available. The following two functions supply the default vernacular and analysis writing systems as a tuple of (language-tag, name):

```
DB.GetDefaultVernacularWS()
DB.GetDefaultAnalysisWS()
```

The language tag is used to specify a non-default writing system for a number of FLEXPProject methods. DB.WSUIName(languageTagOrHandle) will supply the display name for the given language tag.

Text Fields

Fieldworks fields that can contain multiple writing systems have a number of properties for accessing the best text in a given context. For example, "AnalysisDefaultWritingSystem" and "VernacularDefaultWritingSystem" yield the text in the first analysis or first vernacular writing system, respectively. Alternatively, "BestVernacularAlternative" yields the text in the first vernacular writing system that contains data. These are the defined properties:

- AnalysisDefaultWritingSystem;
- VernacularDefaultWritingSystem;
- BestAnalysisAlternative;
- BestVernacularAlternative;
- BestAnalysisVernacularAlternative;
- BestVernacularAnalysisAlternative;

These return an ItsString, so in Python it is necessary to use a cast like this:

```
pronunciation =  
ITsString(p.Form.BestVernacularAlternative).Text
```

To specify a specific writing system use the get_String method with a writing system handle (an integer like 999000004). Use DB.WSHandle(tag) to get the handle for a language tag (e.g. 'en'):

```
ws = DB.WSHandle('por')  
for e in DB.LexiconAllEntries():  
    lexeme = DB.LexiconGetLexemeForm(e)  
    for p in e.PronunciationsOS:  
        report.Info(ITsString(p.Form.get_String(ws)).Text)
```

Custom Fields

Custom fields can be defined in Fieldworks at entry or sense level. The following helper functions are supplied in FLEXPProject to get a list of all the custom fields, or to locate a custom field by name:

```
LexiconGetEntryCustomFields()  
LexiconGetSenseCustomFields()  
fieldID = LexiconGetEntryCustomFieldNamed(fieldName)  
fieldID = LexiconGetSenseCustomFieldNamed(fieldName)
```

The following helper functions are for reading and writing custom fields:

```
LexiconGetFieldText(senseOrEntryOrHvo, fieldID)  
LexiconSetFieldText(senseOrEntryOrHvo, fieldID, text,  
ws=None)
```

Note that the latter will overwrite any formatting in the TsString, setting it all to the given writing system (or the default analysis writing system if ws is not supplied.)

Texts

DB.TextsGetAll(supplyName=True, supplyText=True) returns a list of tuples of (Name, Text) where Text is a string with newlines separating paragraphs. Passing supplyName or supplyText=False returns only the names or texts as a list.

```
for name, text in DB.TextsGetAll():
    report.Info("Processing text: %s" % name)
    numTexts      += 1
    numWords      += len(text.split())
```

Going Deeper

FLEXPProject is not intended to replace LCM, so if you need to do things beyond what is documented here or is shown in the example modules, then you will need to refer to the LCM documentation. Some significant changes were made in Fieldworks version 7, and then in version 8. Most of the documentation hasn't caught up, but it is available [here](#). The relevant documents are:

Python database access.doc ²	Introduction to using Python with FDO/LCM.
Conceptual model overview.doc ³	An explanation of the FieldWorks database structure.

The most up-to-date reference available is the LCMBrowser⁴ application, which can be used to explore a Fieldworks project and see what the fields and relationships are. LCMBrowser can be launched from the FLEXPTools Help menu.

When FLEXPProject helper functions don't meet your needs you can directly access the LCM structures through:

DB.lp	the language project, 'LangProject',
DB.lexDB	the lexical database, 'LexDbOA',
DB.project	the LCM cache object.

Additionally, LCM object repositories can be accessed via the following functions. These take the interface class as the only parameter:

² http://downloads.sil.org/FieldWorks/Documentation/Python_database_access.pdf

³ http://downloads.sil.org/FieldWorks/Documentation/Conceptual_model_overview.pdf

⁴ Installed with Fieldworks.

DB.ObjectsIn(repository) an iterator over all of the objects

DB.ObjectCountFor(repository) the number of objects

```
from SIL.LCModel import ITextRepository

for text in DB.ObjectsIn(ITextRepository):
    report.Info("Text has %d paragraphs" %
                text.ContentsOA.ParagraphsOS.Count)
```

Modifying the Fieldworks Database

The third parameter passed to the processing function is a Boolean that is True if database modifications are allowed (*modifyAllowed*). By default FLEXTools passes in False as a protection against accidental changes. It is only when the user clicks the *Run (Modify)* buttons or holds down the *Shift* key when running a Module, that this parameter will be True.

It is up to the Module author to ensure that the *modifyAllowed* Boolean is adhered to by bracketing all places that modify the database with a conditional on this parameter. (FLEXTools will raise an exception if an attempt is made to change the database, but *modifyAllowed* is False.) In most cases it would be helpful to the user to output different messages depending on the state of this flag (e.g. to say what *would* be changed if *modifyAllowed* was True.)

Additionally the Module documentation value *moduleModifiesDB* must be set to True if the Module can make changes to the database. FLEXTools will never pass through True if *moduleModifiesDB* is False.

Changing Strings

A few functions are provided in FLEXPProject for setting field values, however it is important to be aware of what type of string a certain field is. If it is a MultiString (e.g. Definition and Example) or String then it can embed text in various writing systems, so it is not straight-forward to read and write such fields. The 'Get' functions in FLEXPProject simply return the plain text from MultiString fields. `LexiconSetExample()` writes to the example field, however it will overwrite any formatting that was present.

Flag Field

Where automated changes have risks (like losing string formatting as above) or a correction needs user input, it is helpful to use a custom field as a place to flag the need for attention. `LexiconAddTagToField()` is provided for doing just this. It will append a tag string to the end of the given field inserting semi-colons (;) between tags. If the tag string is already in the field it won't be added a second time. This allows the user to filter on various tags within Fieldworks and then make manual edits to the relevant data.

This example shows how to locate a custom sense-level field called FTFlags, and to update it with error tags using `LexiconAddTagToField()`. FTFlags should be created in Fieldworks as a Sense-level 'Single-line text' field using the 'First Analysis Writing System.' Note that field names are case-sensitive.

```
TestSuite = [ (re.compile(r"[?!\\.]{1}$"), False,
               "ERR:no-ending-punc"),
               (re.compile(r"[?!\\.]{2,}$"), True,
               "ERR:too-much-punc")]

AddReportToField = modify

flagsField = DB.LexiconGetSenseCustomFieldNamed("FTFlags")
if AddReportToField and not flagsField:
    report.Error("FTFlags field missing at Sense level")
    AddReportToField = False

for e in DB.LexiconAllEntries():
    lexeme = DB.LexiconGetLexemeForm(e)
    for sense in e.SensesOS:
        for example in sense.ExamplesOS:
            exText = DB.LexiconGetExample(example)
            if exText == None:
                report.Warning("Blank example: %s" % lexeme,
                              DB.BuildGotoURL(e))
                continue
            for test in TestSuite:
                regex, result, message = test
                if (regex.search(exText) <> None) \
                    == result:
                    report.Warning("%s: %s" % (lexeme,
                                                message),
                                  DB.BuildGotoURL(e))
            if AddReportToField:
                DB.LexiconAddTagToField(sense,
                                         flagsField,
                                         message)
```

(See Examples\Example_Check_Punctuation.py)

Documenting the Module

Module documentation is supplied in a Python dictionary as follows:

```
docs = {FTM_Name      : "Demo",
        FTM_Version   : 1,
        FTM_ModifiesDB : False,
        FTM_Synopsis    : "Does nothing",
        FTM_Help       : "Demo help.htm",
        FTM_Description: ""}
```

FTM_Name	A descriptive name for the module.
FTM_Version	An integer or string version identifier.
FTM_ModifiesDB	True if the module modifies the database.
FTM_Synopsis	A brief one-sentence description of the module.
FTM_Help	The name of a help file for this module. The path is relative to the current directory. The help file should be HTML or PDF for portability. Define as None if there is no help file.
FTM_Description	A full description of the module's function, purpose, constraints, etc. This can be a multi-line string using triple-quotes (""").

Debugging

Syntax errors and import errors are reported when FLEXTools starts up. Such errors do not prevent FLEXTools from running, but any Module with an error cannot be run until the error is fixed. Use the menu FLEXTools | Re-load Module to re-import all Modules after making any edits.

Run-time exceptions are reported in the message window with the details in the tool tip.

Run Stand-alone

If there are problems such as FLEXTools failing to start with no error messages then it may be helpful to run the module in stand-alone mode. You can do this with the `_TestAModule.py` script: Edit it to load the module to be tested and specify the database to use. Open a command window in the FlexTools folder and run:

```
py _TestAModule.py
```

Extra Debugging Output

Extra report function calls can be used to output debugging information, or if you want to separate debugging messages from the normal output you can use logging statements and view flextools.log after FLExTools has been closed.

Configure logging with:

```
import logging
logger = logging.getLogger(__name__)
```

Log entries are made as follows:

```
logger.info("general informative message")
logger.error("an error message")
logger.debug("this is only output in DEBUG mode")
```

Use FlexTools_Debug.bat to run in DEBUG mode.