FLExTools Programming Help

For FLExTools Version 2.3,
and FieldWorks version 9.1 & 9.2.
5 June 2025

Contents

Introduction

FLExTools is a framework for running Python scripts on a FieldWorks Language Explorer project. FLExTools provides core functions for creating *modules* (Python scripts) 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 flextoolslib import *

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

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

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

A `FlexToolsModuleClass` is initialised with the main processing function (*Demo* in this example) and the documentation. The instance must be called `FlexToolsModule`.

Notice the three parameters to the *Demo* function:

- `project`—this a `FLExProject` object, giving access to the LCM[1] cache and helper functions for reading from and writing to the project.
- `report`—message reporting functions and progress indication.
- `modifyAllowed`—a flag to tell the module whether the user wants to do a dry-run (`modifyAllowed==False`), or make changes to the project (`modifyAllowed==True`).

[1]Language and Culture Model; formerly called FieldWorks Data Objects (FDO)

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

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 project.
`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 (""" """).

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:

`report.Info(msg, info)`	A basic information message.
`report.Warning(msg, info)`	A non-critical warning such as telling the user about data changes or integrity issues.
`report.Error(msg, info)`	A critical situation that prevents further processing.
`report.Blank()`	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 project you can pass

`project.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:

`report.ProgressStart(max)`	`Start the progress indicator, setting the maximum value to` *`max.`*
`report.ProgressUpdate(value)`	`Update the progress indicator to` *`value.`* `(`*`value`*` = [0...max-1])`

`For example:`

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

FieldWorks project functions

The FieldWorks project is accessed via the `project` parameter to the module's runFunction. `project` is an instance of the `FLExProject` class. The class definition documentation can be found on the menu Help | API Help. This class contains helper functions for accessing the FieldWorks project via the FieldWorks LCM interface.

Basic project functions

```
for e in project.LexiconAllEntries():
    for sense in e.SensesOS:
        if not project.LexiconGetSenseDefinition(sense):
            report.Warning(f"Missing definition in
                        {project.LexiconGetLexemeForm(e)}",
                          project.BuildGotoURL(e))
        if len(sense.ExamplesOS) == 0:
            report.Warning(f"No examples in
                        {project.LexiconGetLexemeForm(e)}",
                          project.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 project.LexiconAllEntries()[:20]:
```

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

```
project.LexiconGetSenseGloss(sense [, languageTagOrHandle])
project.LexiconGetSensePOS(sense)
```

```
project.LexiconGetSenseSemanticDomains(sense)
```

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

```
for ex in sense.ExamplesOS:
  Report.Info(f"Example: {project.LexiconGetExample(ex)}")
```

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

Advanced lexicon operations

The above example traverses the lexicon in an unspecified 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 project.LexiconAllEntries():
    # Add fields of interest to 'data'
    data = {}
    data['hvo'] = e.Hvo
    glosses = [project.LexiconGetSenseGloss(s) for s in
e.SensesOS]
    data['glosses'] = glosses
    # The key is a 2-tuple: (lexeme-form, homograph-number)
    LexiconEntries[(project.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 project, showing the display name, the language tag (e.g. `'en'`), and writing system handle.

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

```
project.GetDefaultVernacularWS()
project.GetDefaultAnalysisWS()
```

The language tag is used to specify a non-default writing system for a number of FLExProject methods. `project.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 `project.WSHandle(tag)` to get the handle for a language tag (e.g. `'por'`):

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

Unicode

FieldWorks stores all of its text data in decomposed normal form (NFD)[2]. When dealing with data that may contain diacritics or other combining forms, it is necessary to convert to NFD for comparing or writing text strings. This can be done with the `unicodedata.normalize()` function:

```
pinyin    = project.ReversalGetForm(entry, ChinesePinyinWS)
newPinyin = TonenumberToPinyin(tonenum)
newPinyin = unicodedata.normalize('NFD', newPinyin)
if newPinyin != pinyin:
    project.ReversalSetForm(entry, newPinyin, ChinesePinyinWS)
```

[2] https://en.wikipedia.org/wiki/Unicode_equivalence#Combining_and_precomposed_characters

Custom fields

Custom fields can be defined in FieldWorks at entry or sense level. The following helper functions are supplied in `FLExProject` 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

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

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

Going deeper

`FLExProject` 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 FieldWorks technical documentation. The relevant documents are:

FLEx 9.1 Conceptual Model[3]	An explanation of the FieldWorks database structure.
FieldWorks model diagrams[4]	Class definitions and diagrams for the FieldWorks conceptual model. This dates to FieldWorks 7.2, so a number of things are out of date. (Note: If using Windows 10 or 11, right-click the downloaded file, choose Properties and check the 'unblock' box to be able to open the file.)
Model 7000072 classes and fields[5]	A spreadsheet with all the class and field

[3] https://downloads.languagetechnology.org/fieldworks/Documentation/FLEx%209.1%20Conceptual%20Model.pdf

[4] https://downloads.languagetechnology.org/fieldworks/Documentation/ModelDocumentation.chm

	information. This is the latest model definition.
Interlinear text implementation in FLEx[6]	Describes the technical details of interlinear texts in FieldWorks 9.
Python for FlexTools and FLEx 9.1[7]	An introduction to using Python with the LCM.

Another model reference is the LCMBrowser[8] application. This can be used to explore a FieldWorks project and see what the fields and relationships are. LCMBrowser can be launched from the FLExTools Help menu.

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

`project.project`	the LCM cache object
`project.lp`	the language project: `project.LangProject`,
`project.lexDB`	the lexical database: `project.LangProject.LexDbOA`

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

`project.ObjectsIn(repository)`	an iterator over all of the objects
`project.ObjectCountFor(repository)`	the number of objects

For example:

```
from SIL.LCModel import ITextRepository

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

Modifying the FieldWorks project

The third parameter, `modifyAllowed`, passed to the processing function is a Boolean that is True if project modifications are allowed. By default, FLExTools passes 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.

---

[5] https://downloads.languagetechnology.org/fieldworks/Documentation/MasterFieldWorksModel%20classes%20and%20fields%207000072.xlsx

[6] https://downloads.languagetechnology.org/fieldworks/Documentation/Technical_Notes_on_Interlinear_Import.pdf

[7] https://downloads.languagetechnology.org/fieldworks/Documentation/Python%20for%20FlexTools%20and%20FLEx%209.1.pdf

[8] Installed with FieldWorks.

It is up to the module author to ensure that the `modifyAllowed` Boolean is adhered to by bracketing all places that modify the project with a conditional on this parameter. (FLExTools will raise an exception if an attempt is made to change the project, 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 `FTM_ModifiesDB` must be set to True if the module can make changes to the project. FLExTools will never pass through True if `FTM_ModifiesDB` is False.

Changing strings

A few functions are provided in FLExProject 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 `FLExProject` 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 and insert 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 the 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 = modifyAllowed

flagsField =
      project.LexiconGetSenseCustomFieldNamed("FTFlags")
if AddReportToField and not flagsField:
    report.Error("The Sense-level FTFlags field is missing")
    AddReportToField = False

for e in project.LexiconAllEntries():
```

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

(See Examples\Example_Check_Punctuation.py)

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 or F5 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 *scripts\TestAModule.py* script: Open a command window in the FlexTools\scripts folder and run:

```
py TestAModulule.py ..\Modules\<Folder>\<Module> <Project>
```

Extra debugging output

The Python repr function converts any Python object into a string represenation, which can be used with report.Info() to provide extra debugging information. For example:

```
report.Info(repr(POSList))
```

To separate debugging messages from the normal output logging statements can be used and the flextools.log viewed 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.vbs* to run in DEBUG mode.

For more serious debugging, use PyCharm[9], Visual Studio[10] or any other IDE that supports Python.

[9] https://www.jetbrains.com/pycharm/

[10] https://visualstudio.microsoft.com/free-developer-offers/