OpenFormats Documentation

Release 0.1

Transifex

November 22, 2016

Contents

OpenFormats is a localization file format library, written in Python.

- Read and write to various file formats such as *.po*, *.xliff* or even ones which are not localization formats, such as *.srt* and *.txt*.
- Plural support for the formats which do support it.
- Built-in web-based test app, to help you develop your own format handlers.

OpenFormats' primary use is to work as a file format backend to Transifex.

Check out OpenFormats documentation for more information.

CHAPTER 1

How to get help, contribute, or provide feedback

See our contribution submission and feedback guidelines.

You can run tests for the formats by doing the following:

```
python setup.py test
```

CHAPTER 2

Source code

The source code for OpenFormats is hosted on GitHub.

CHAPTER 3

The testbed

To run the testbed:

```
./manage.py syncdb --noinput  # optional
./manage.py runserver
```

Then point your browser to http://localhost:8000/.

The *syncdb* step is optional and is used if you wish to save certain tests by their URL The tests are saved to an sqlite database. This is most likely to be useful in the live version of the testbed.

Having fired up the testbed, you can select a format handler, paste some text and try to parse it. The testbed will show you the stringset that was extracted from the source text and the template in kept from it. Then, you can try compiling the template against the stringset, or you can modify it first.

CHAPTER 4

Contents

4.1 Why all this fuss?

This library performs one of the most important functions of Transifex: The use of language files to import and deliver translations.

4.1.1 How software localization works (in a nutshell)

Your software stack comes with a tool (if it doesn't it should) that finds all translatable text in your product and extracts it to a language file. We'll call this the **source language file**, which is placed in a special folder in your product's code.

Once you get that in place, your job is to produce several **target language files**, once for each language you want your product to appear in and place them in the same folder. These language files are very similar to the source language file, all that changes is that in the same place your source strings would be, there are now translations.

Your software stack will be able to pull translations from the language files and put them in place of the original strings in your product, if the user chooses a translated language. Tada!!!

Here is a sample source language file:

```
# Translation file for Transifex.
# Copyright (C) 2007-2010 Indifex Ltd.
# This file is distributed under the same license as the Transifex package.
msgid ""
msgstr ""
"Project-Id-Version: Transifex\n"
"POT-Creation-Date: 2012-09-27 09:17+0000\n"
"PO-Revision-Date: 2012-09-27 10:07+0000\n"
"Last-Translator: Ilias-Dimitrios Vrachnis <vid@transifex.com>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Language: en\n"
"Plural-Forms: nplurals=2; plural=(n != 1);\n"

#: accounts/forms.py:22 accounts/forms.py:193
msgid "Username"
msgstr "Username"

#: accounts/forms.py:24 accounts/forms.py:195
msgid "Username must contain only letters, numbers, dots and underscores."
msgstr "Username must contain only letters, numbers, dots and underscores."
```

```
#: accounts/forms.py:27 accounts/forms.py:182 accounts/forms.py:198
msgid "Email"
msgstr "Email"
```

And here is a sample target language file:

```
# Translation file for Transifex.
# Copyright (C) 2007-2010 Indifex Ltd.
# This file is distributed under the same license as the Transifex package.
msgid ""
msgstr ""
"Project-Id-Version: Transifex\n"
"POT-Creation-Date: 2012-09-27 09:17+0000\n"
"PO-Revision-Date: 2015-05-26 21:35+0000\n"
"Last-Translator: Kadministrator Bairaktaris <kb_admin@kbairak.com>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"Language: el\n"
"Plural-Forms: nplurals=2; plural=(n != 1);\n"

#: accounts/forms.py:22 accounts/forms.py:193
msgid "Username"
msgstr "νομα χρστη"

#: accounts/forms.py:24 accounts/forms.py:195
msgid "Username must contain only letters, numbers, dots and underscores."
msgstr "Το νομα χρστη πρπει να περιχει μνο γρμματα, αριθμο, τελεε και κτω παλε."

#: accounts/forms.py:27 accounts/forms.py:182 accounts/forms.py:198
msgid "Email"
msgstr "Διεθυνση ηλεκτρονικο ταχυδρομεου"
```

4.1.2 File formats

As you can see, these language files have a peculiar format. These ones in particular follow the PO file format, and are generated and parsed by an open-source software called gettext, which is popular in the open-source world. The structure of these files allows compatible software to use their contents to display the product in a variety of languages.

We need to support a variety of such file formats, as well of some formats that weren't necessarily made for localization. For example, why shouldn't you be able to use this process to localize subtitle files when the same process can clearly work for those too?

Source:

```
1
00:01:45,105 --> 00:01:47,940
Pinky: Gee, Brain, what do you want to do tonight?

2
00:02:45,105 --> 00:02:47,940
Brain: The same thing we do every night, Pinky - try to take over the world!
```

Translated:

```
1
00:01:45,105 --> 00:01:47,940
```

```
Pinky: Τι θε να κνουμε απψε Brain?

2
00:02:45,105 --> 00:02:47,940
Brain: ,τι κνουμε κθε βρδυ, Pinky - θα προσπαθσουμε να καταλβουμε τον κσμο!
```

4.1.3 How Transifex and Openformats deal with this task

A **handler**, the basic unit of the Openformats library, will parse a source language file and find the source strings in it. It will extract these into a **stringset**, a collection of said content associated with some metadata. This metadata's use is to:

1. Identify the strings and their translations inside the language files
2. Provide context for the translators

The source strings in the source file are replaced by **hashes**, constructed by the metadata we just mentioned. The result of this process is what we call the **template**.

Both the stringset and the template are stored in Transifex's database. The translation editor will present the stringset to translators, abstracting the template away, allowing them to focus solely on translation. Translators in Transifex's web editor can work on a variety of files using the exact same interface, not having to bother with the nature or the structure of the file format being used.

Having saved the trasnlations in the database, the format handler can combine those with the template to produce a target language file to be used in your product. This process is called **compiling**. The handler searches for hashes in the template, associates them with their relevant translation entries using the metadata we stored during parsing and replaces the hashes with the translations. The result is a target language file, ready to be used in your product.

4.1.4 Step-by-step

Lets take the first subtitle from our previous example:

```
1
00:01:45,105 --> 00:01:47,940
Pinky: Gee, Brain, what do you want to do tonight?
```

Here, we need to find the source string and something that will allow us to identify its position later when we want to compile a language file. The string is obviously "Pinky: Gee, Brain, what do you want to do tonight?". For our metadata, we will use the ascending number on top, the '1', since we're guaranteed that it is unique within the source file; if it isn't, our parser should raise an error.

Hashing the identifier (the '1') will give us this: '3afcdbfeb6ecfbdd0ba628696e3cc163_tr'. This is what we will replace our source string with:

```
1
00:01:45,105 --> 00:01:47,940
3afcdbfeb6ecfbdd0ba628696e3cc163_tr
```

This is the template!

In the web editor, the translators will produce a translated string based on our source string:

Language	Text
English	Pinky: Gee, Brain, what do you want to do tonight?
Greek	Pinky: Τι θε να κνουμε απψε Brain?

And, finally, the compiler will be able to find the hash in the template and replace it with the translation:

```
1
00:01:45,105 --> 00:01:47,940
Pinky: Τι θε να κνουμε απψε Brain?
```

4.2 Getting Started Guide

Here are some quick steps to get you started with OpenFormats.

4.2.1 Installation

To use OpenFormats as a Python library, simply install it with `pip`, prefixing with `sudo` if permissions warrant:

```
pip install openformats
```

If you plan to tweak the codebase or add your own format handler, grab a copy of the whole repository from GitHub:

```
git clone https://github.com/transifex/openformats.git
cd openformats
```

4.2.2 Creating your own handler

OpenFormats supports a variety of file formats, including plaintext (`.txt`), subtitles (`.srt`) and others. Here are the steps to create your own handler.

4.2.3 1. Subclass the base *Handler*

class `openformats.handlers.`**Handler**

This class defines the interface you need to implement in order to create a handler. Both the *parse* and *compile* methods must be implemented.

parse(*content*, *is_source=False*)

Parses the content, extracts translatable strings into a stringset, replaces them with hashes and returns a tuple of the template with the stringset

Typically this is done in the following way:

- Use a library or your own code to segment (deserialize) the content into translatable entities.
- Choose a key to uniquely identify the entity.
- Create an `OpenString` object representing the entity.
- Create a hash to replace the original content with.
- Create a stringset with the content.
- Use library or own code to serialize stringset back into a template.

compile(*template*, *stringset*)

Parses the template, finds the hashes, replaces them with strings from the stringset and returns the compiled file. If a hash in the template isn't found in the stringset, it's a good practice to remove the whole string section surrounding it

Typically this is done in the following way:

- Use a library or own code to segment (deserialize) the template into translatable entities, as if assuming that the hashes are the translatable entities.
- Make sure the hash matches the first string in the stringset.
- Replace the hash with the string.
- Use library or own code to serialize stringset back into a compiled file.

You can safely assume that the stringset will have strings in the correct order for the above process and thus you will probably be able to perform the whole compilation in a single pass.

The following are some classes that will help you with this process:

4.2.4 2. The *OpenString* class

class `openformats.strings.`**`OpenString`**(*key*, *string_or_strings*, ***kwargs*)

This class will abstract away the business of generating hashes out of your strings and will serve as a place to get translations from when compiling. Several OpenStrings in our process define a *Stringset*, which is simply a python list of OpenStrings. To create an OpenString, you need 2 arguments:

- The 'key'

  Something in your source file that uniquely identifies the section that the source string originated from. It might be helpful for your compiler to use something that appears in the same form in language files as well.

- The 'string' or 'plural forms of the string':

  If the file format you're working with does not support plural forms, or if the string in question is not pluralized, you can just supply the string itself as the second argument. If you string is pluralized however, you have to supply all plural forms in a dictionary with the rule numbers as keys. For example:

```
OpenString("UNREAD MESSAGES",
           {1: "You have %s unread message",
            5: "You have %s unread messages"})
```

- There are a number of optional keyword arguments to *OpenString*:

  context, order, character_limit, occurrences, developer_comment, flags, fuzzy, obsolete

Their main purpose is to provide context to the translators so that they can achieve higher quality. Two of them however, though optional, are highly recommended:

- Context

  This is also taken into account when producing the hash, so if you can't ensure that your keys aren't unique within the source file, you can still get away with ensuring that the *(key, context)* pair is.

- Order

  If you provide an order (integer), Transifex will save it in the database and then, when you try to compile a template against a stringset fetched from Transifex, it will already be ordered, even if it contains translations. This can allow you to optimize the compilation process as the order that the hashes appear in the template will be the same as the order of strings in the stringset.

  Another valuable outcome is that the order will be preserved when the strings are shown to translators which can provide context and thus improve translation quality.

Once you have created an OpenString, you can get it's hash using the *template_replacement* property

4.2.5 3. The *Transcriber*

class openformats.transcribers.**Transcriber**(*source*)

This class helps with creating a template from an imported file or compile an output file from a template.

Main functionality

This class will help with both creating a template from an imported file and with compiling a file from a template. It provides functions for copying text. It depends on 3 things, the source content (self.source), the target content (self.destination) which initially will contain an empty string and a pointer (self.ptr) which will indicate which parts of 'source' have already been copied to 'destination' (and will be initialized to 0).

Transcriber detects and remembers the newline type (DOS, `'\r\n'` or UNIX `'\n'`) of 'source'. It then converts 'source' to UNIX-like newlines and works on this. When returning the destination, the initial newline type will be used. Because 'source' is being potentially edited, it's a good idea to save Transcriber's source back on top of the original one:

```
>>> def parse(self, source):
...     self.transcriber = Transcriber(source)
...     self.source = self.transcriber.source
...     # ...
```

The main methods provided are demonstrated below:

```
>>> transcriber = Transcriber(source)

source:      <string name="foo">hello world</string>
ptr:         ^ (0)
destination: []

>>> transcriber.copy_until(source.index('>') + 1)

source:      <string name="foo">hello world</string>
ptr:                           ^
destination: ['<string name="foo">']

>>> transcriber.add("aee8cc2abd5abd5a87cd784be_tr")

source:      <string name="foo">hello world</string>
ptr:                           ^
destination: ['<string name="foo">', 'aee8cc2abd5abd5a87cd784be_tr']

>>> transcriber.skip(len("hello world"))

source:      <string name="foo">hello world</string>
ptr:                                     ^
destination: ['<string name="foo">', 'aee8cc2abd5abd5a87cd784be_tr']

>>> transcriber.copy_until(source.index("</string>") +
...                         len("</string>"))

source:      <string name="foo">hello world</string>
ptr:                                                ^
destination: ['<string name="foo">', 'aee8cc2abd5abd5a87cd784be_tr',
'</string>']

>>> print transcriber.get_destination()

<string name="foo">aee8cc2abd5abd5a87cd784be_tr</string>
```

remove_section (*place=0*)

You can mark sections in the target file and optionally remove them. Insert the section-start and section-end bookmarks wherever you want to mark a section. Then you can remove a section with *remove_section()*. For example:

```
>>> transcriber = Transcriber(source)

source:      <keep><remove>
ptr:         ^ (0)
destination: []

>>> start = 0

>>> transcriber.mark_section_start()
>>> transcriber.copy_until(start + 1)  # copy until first '<'
>>> string = source[start + 1:source.index('>', start)]
>>> transcriber.add("asdf")  # add the hash
>>> transcriber.skip(len(string))
>>> transcriber.copy_until(source.index('>', start) + 1)
>>> transcriber.mark_section_end()

source:      <keep><remove>
ptr:               ^
destination: [SectionStart, '<', 'asdf', '>', SectionEnd]

>>> if string == "remove":
...     transcriber.remove_section()

(nothing happens)

>>> start = source.index('>') + 1

>>> # Same deal as before, mostly
>>> transcriber.mark_section_start()
>>> transcriber.copy_until(start + 1)  # copy until second '<'
>>> string = source[start + 1:source.index('>', start)]
>>> transcriber.add("fdsa")  # add the hash
>>> transcriber.skip(len(string))
>>> transcriber.copy_until(source.index('>', start) + 1)
>>> transcriber.mark_section_end()

source:      <keep><remove>
ptr:                        ^
destination: [SectionStart, '<', 'asdf', '>', SectionEnd,
              SectionStart, '<', 'fdsa', '>', SectionEnd]

>>> if string == "remove":
...     transcriber.remove_section()

source:      <keep><remove>
ptr:                        ^
destination: [SectionStart,  '<', 'asdf', '>',  SectionEnd,
              None        , None, None  , None, None      ]

(The last section was replaced with Nones)

Now, when you try to get the result with `get_destination()`, the
Nones, SectionStarts and SectionEnds will be ommited:
```

```
        >>> transcriber.get_destination()

        <asdf>
```

line_number
The transcriber remembers how many newlines it has went over on the source, both when copying and skipping content. This allows you to pinpoint the line-number a parse-error has occured. For example:

```
        source:
            first line
            second line
            third line with error
            fourth line

        >>> transcriber = Transcriber(source)
        >>> for line in source.split("\n"):
        >>>     if "error" not in line:
        >>>         # include the newline too
        >>>         transcriber.copy(len(line) + 1)
        >>>     else:
        >>>         raise ParseError(
        >>>             "Error on line {line_no}: '{line}'".format(
        >>>                 line_no=transcriber.line_number,
        >>>                 line=line
        >>>             )
        >>>         )

        This will raise a::

        >>> ParseError("Error on line 3: 'third line with error'")
```

edit_newlines(*chunk*, *enforce_newline_type=None*)
This is the part that renders the newlines to their correct type when returning the final result. You have the option to enforce the newline type if you want to.

```
        >>> source = "hello\r\nworld"
        >>> t = Transcriber(source)
        >>> t.source
```

```
        >>> "hello\nworld"
```

```
        >>> source = trascriber.source
        >>> # Work as if source was UNIX-type
        >>> t.copy_until(source.index('\n') + 1)  # include the '\n'
        >>> t.add("fellas")
        >>> t.get_destination()
```

```
        >>> "hello\r\nfellas"  # <- it remembered newline type from source
```

```
        >>> t.get_destination(enforce_newline_type="UNIX")
```

```
        >>> "hello\nfellas"
```

Continue reading the other documentation sections for further details.

4.3 Testing

4.3.1 1. Get yourself a sample file

Its a very good idea to start developing by getting a sample source file for two reasons:

1. It will get picked up by the *The Testbed* so you will be able to get instant feedback as you work on your handler.
2. You will get a lot of tests for free. These tests will parse the sample file into a template and stringset, compile them back in your source file and check whether the template matches the expected one and that the resulted file matches the source. It will also try to translate the strings based on some common ones found in a dictionary and check that it can compile a language file that matches the expected one.

Put your sample file in `openformats/tests/formats/<format_name>/files/1_en.<format_extension>`. For example, our sample SRT file goes to `OpenFormats/tests/formats/srt/files/1_en.srt`.

4.3.2 2. Generate expected template and language files

In order to generate the expected template and language files mentioned above, you can use the *bin/create_files.py* script once you have a working handler:

```
./bin/create_files.py openformats/tests/formats/srt/files/1_en.srt
```

In order to get the tests we mentioned for free, make sure your test class inherits from the:

class `openformats.tests.formats.common.`**`CommonFormatTestMixin`**

Define a set of tests to be run by every file format.

The class that inherits from this must define the following:

- `HANDLER_CLASS`, eg: PlaintextHandler
- `TESTFILE_BASE`, eg: *openformats/tests/formats/plaintext/files*

You might have noticed that by using a working handler to make the expected sample files and then testing against them seems pointless. Well, you're right, they are, initially. The point of for them to serve as regression tests, as you later make changes to your handler.

4.3.3 3. Add your own tests

Testing that a handler works correctly against a valid source file is good, but you will want to also test more things, like:

- The hashes produced take the correct information into account
- The metadata of the extracted strings is what you want
- *ParseErrors* are raised when appropriate and they produce a helpful error message
- Sections of the compiled files are removed when the relevant strings are missing from the stringset given as input
- Anything to get your coverage higher

4.3.4 4. Utilities

openformats.tests.utils.**generate_random_string**(*length=20*)

openformats.tests.utils.**strip_leading_spaces**(*source*)

This is to help you write multilingual strings as test inputs in your tests without screwing up your code's syntax. Eg:

```
    '''
        1
        00:01:28.797 --> 00:01:30.297 X:240 Y:480
        Hello world
    '''
```

will be converted to:

```
    '\n1\n00:01:28.797 --> 00:01:30.297 X:240 Y:480\nHello world\n'
```

CommonFormatTestMixin.**_test_parse_error**(*source*, *error_msg*, *parse_kwargs=None*)

Test that trying to parse 'source' raises an error with a message exactly like 'error_msg'

4.3.5 5. Run the test suite

```
python setup.py test
```

4.4 The Testbed

The testbed is a real-time django web application included with the openformats library to help you develop, test and debug format handlers. To start it, simply run:

```
./manage.py runserver
```

and point your browser to `http://localhost:8000`

The interface consists of 3 columns, one for each state of the handler's lifetime.

4.4.1 The source column

From here you choose which file format you want to play around with. The list is automatically populated by all sublasses of *Handler* defined in all modules in `openformats/formats`.

Once you select one, you can type or paste some content in the textarea. If you have a sample file in `openformats/tests/formats/<format_name>/files/1_en.<extension>`, it will be picked up by the testbed and put in the textarea automatically once you select a format. You can of course edit or replace it if you want.

Finally, press 'parse' to, well, parse the source content. The actual handler will be used to display the outcome in the next column:

4.4.2 The stringset-template column

This column shows the outcome of the previous operation: the stringset and template extracted from the source. You can inspect entries of the stringset, edit their content or even delete them. Once you're ready, you can press on compile to have the handler create a language file out of the template and the, potentially edited, stringset.

4.4.3 The compiled column

This shows the outcome of the previous operation. There is also a message that tells you if the compiled text matches the source, in case you didn't edit the stringset and this is what you had expected.

4.4.4 Errors

If there's an error during the parsing or compiling operation, a full traceback will be printed on the relevant column. This is helpful for both debugging and making sure that the error messages displayed to the user when there is a mistake in the source file is accurate and helpful.

4.4.5 Saving tests

If you run the following command:

```
./manage.py syncdb --noinput
```

The testbed will be able to save your current test state (chosen format, soruce, stringset, template, compiled file) in an sqlite database and allow you to play it back any time. This saved test can be accessed from the URL in your browser right after you've pressed the save button.

You will probably not need to do that yourself; this is a feature intended for the public hosted version of the testbed, so that users can provide Transifex support or openformats contributors with test cases that reproduce a bug.

4.5 Utils

4.5.1 Compilers

class openformats.utils.compilers.**OrderedCompilerMixin**
Bases: object

4.6 Contributing to OpenFormats

4.6.1 Filing issues

- Before you file an issue, try *asking for help* first.
- If determined to file an issue, first check for existing issues, including closed issues.

4.6.2 How to get help

Before you ask for help, please make sure you do the following:

1. Read the documentation thoroughly. If in a hurry, at least use the search field that is provided on the documentation pages.
2. Use a search engine (e.g., DuckDuckGo, Google) to search for a solution to your problem. Someone may have already found a solution.
3. Try reproducing the issue in a clean environment, ensuring you are using:

- correct latest OpenFormats release (or an up-to-date git clone of master)
- latest releases of libraries used by Openformats
- no plugins or only those related to the issue

If despite the above efforts you still cannot resolve your problem, be sure to include in your inquiry the following information, preferably in the form of links to content uploaded to a paste service, GitHub repository, or other publicly-accessible location:

- Describe what version of Openformats you are running or the HEAD commit hash if you cloned the repo) and how exactly you installed it (the full command you used, e.g. `pip install openformats`).
- If you are looking for a way to get some end result, prepare a detailed description of what the end result should look like (preferably in the form of an image or a mock-up page) and explain in detail what you have done so far to achieve it.
- If you are trying to solve some issue, prepare a detailed description of how to reproduce the problem. If the issue cannot be easily reproduced, it cannot be debugged by developers or volunteers. Describe only the **minimum steps** necessary to reproduce it (no extra plugins, etc.).
- Upload any settings file or any other custom code that would enable people to reproduce the problem or to see what you have already tried to achieve the desired end result.
- Upload detailed and **complete** output logs and backtraces.

Once the above preparation is ready, you can contact people willing to help via a GitHub issue or send a message to `support at transifex dot com`. Remember to include all the information you prepared.

4.6.3 Contributing code

Before you submit a contribution, please ask whether it is desired so that you don't spend a lot of time working on something that would be rejected for a known reason.

Using Git and GitHub

- Create a new git branch specific to your change (as opposed to making your commits in the master branch).
- **Don't put multiple unrelated fixes/features in the same branch / pull request.** For example, if you're hacking on a new feature and find a bugfix that doesn't *require* your new feature, **make a new distinct branch and pull request** for the bugfix.
- Check for unnecessary whitespace via `git diff --check` before committing.
- First line of your commit message should start with present-tense verb, be 50 characters or less, and include the relevant issue number(s) if applicable. *Example:* `Ensure proper PLUGIN_PATH behavior. Refs #428.` If the commit *completely fixes* an existing bug report, please use `Fixes #585` or `Fix #585` syntax (so the relevant issue is automatically closed upon PR merge).
- After the first line of the commit message, add a blank line and then a more detailed explanation (when relevant).
- Squash your commits to eliminate merge commits and ensure a clean and readable commit history.
- If you have previously filed a GitHub issue and want to contribute code that addresses that issue, **please use** `hub pull-request` instead of using GitHub's web UI to submit the pull request. This isn't an absolute requirement, but makes the maintainers' lives much easier! Specifically: install hub and then run hub pull-request to turn your GitHub issue into a pull request containing your code.

Contribution quality standards

- Adhere to PEP8 coding standards whenever possible. This can be eased via the pep8 or flake8 tools, the latter of which in particular will give you some useful hints about ways in which the code/formatting can be improved.
- Add docs and tests for your changes. Undocumented and untested features will not be accepted.
- *Run all the tests* **on all versions of Python supported by Openformats** to ensure nothing was accidentally broken.

Check out our Git Tips page or *ask for help* if you need assistance or have any questions about these guidelines.

4.7 Changelog

4.7.1 0.1 (2015-05-15)

Initial release.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

O

Symbols

C

E

G

H

L

O

P

R

S

T