Poradnik

Tłumaczenia wtyczek GIMP, czyli jak wygenerować plik .pot korzystając z PoEdit.

05-10-2015r

Wprowadzenie

Niestety nie każdy jest w stanie zrozumieć wtyczkę po angielsku.
Może kiedyś zastanawiałeś się dlaczego niektóre wtyczki zaraz po zainstalowaniu "*przemówiły*" po polsku (choć autor był powiedzmy Niemcem, a wtyczkę przygotował w języku angielskim), jeśli kiedykolwiek zauważyłeś w katalogu wtyczki pliki z rozszerzeniem **.po** i **.mo** i zawierające w swojej nazwie pl_PL, jesteś już na początku drogi do prawidłowej internacjonalizacji posiadanych wtyczek, aby używać GIMP-a bez korzystania ze słownika, aby odczytać nieznane słowa.

Dla rozwiązania tego problemu możemy podejść dwoma drogami.
Po pierwsze – *lokalizacja*, czyli przetłumaczenie na język ojczysty tylko odpowiednich fraz (**ciągów**) wtyczki, co w gruncie rzeczy nie jest zbytnim utrudnieniem *oraz*
Po drugie – *Internacjonalizacja* wtyczki, czyli stworzyć kolejną wersję lokalizacji korzystając np. z **PoEdit** i pliku **.pot** czy **.po**. W tym celu czasami autorzy wtyczek dostarczają plik **.po**, a nawet **.pot**.

Ale co zrobić, jeśli chcemy lub trzeba przetłumaczyć wtyczkę na Nasz język, a tych plików brak?
Czy może szukałeś tego pliku **.po** w katalogu wtyczki, aby tylko stwierdzić, że go tam nie ma?
A jeśli jest to prawda, to co można zrobić?
Podam poprawny i profesjonalny sposób na to, aby wygenerować plik **.pot**. z pliku wtyczki **.py** i dalej **.po**
W finałowym stadium pliki **.po** zamieniane będą na binarny (wykonywalny) format **.mo**.
Plik **pl.po**, tłumaczący można edytować ręcznie lub przy użyciu narzędzi takich jak **Poedit**, **Emacs** lub **Vim**, które posiadają tryb edycji plików tłumaczeń gettext.

Szkoda, że na **stronach GIMP-a**, nie znalazłem na ten temat żadnych informacji.
Mam nadzieję, że pokażę prawidłowy sposób, tak, aby każdy mógł bez problemu wygenerować plik **.pot**.
Poradnik przeznaczony jest głównie do stosowania przez tłumaczy amatorów.
Założenie:

Większość dystrybucji GIMP-a na dzień dobry ma zainstalowanego Pythona z dodatkowi bibliotekami. W tych dystrybucjach Python-Fu dostępny jest od początku.

Oficjalny GIMP instalator dla systemu Windows bezpośrednio z download.gimp.org, zawiera zarówno wersje 32-bitowe i 64-bitowe GIMP i zainstaluje odpowiednią.

Aby coś zrozumieć, potrzebne jest trochę terminologii:

Razem z instalacją Pythona uzyskujemy dostęp do mechanizmów biblioteki **gettext** osiąganych przy użyciu zawartego w Pythonie *modułu* **gettext,** który zapewnia wsparcie dla wielojęzycznych tłumaczeń, a mianowicie Internacjonalizację **(i18n**) i usługi lokalizacyjne **(l10n)** dla modułów Pythona i aplikacji.
Nie należy mylić **gettext** jako takiego (zewnętrznego w odniesieniu do Pythona, który nie jest wymagany do jego pracy;), tym nie mniej, jego komplet zawiera wygodne narzędzia do pracy z plikami gettext po wbudowaniu w Pythona.
Biblioteka **gettext** służy do tłumaczenia programów w wielu różnych językach nie tylko w Pythonie.
Pozwala ona używać w naszym programie szablonów zwrotów, które mogą być tłumaczone za pomocą odrębnych i niezależnych plików tłumaczenia. Biblioteka **gettext** pozwala pobrać przetłumaczone ciągi, z pliku tłumaczeń, i przy istnieniu odpowiedniego dla lokalnego języka pliku tłumaczenia, zastąpić niezbędny ciąg.
Podstawowa literatura: http://www.gnu.org/software/gettext/manual/gettext.html !!!
Plug-in Internationalization http://gimp-plug-ins.sourceforge.net/doc/i18n.html
http://www.gtk.org/api/2.6/glib/glib-I18N.html#N-%3aCAPS
http://wiki.wxpython.org/Internationalization
Najpopularniejsza implementacja **gettext** jest rozwijana przez Projekt GNU w ramach wolnego oprogramowania. Wersja stworzona przez Projekt GNU została wydana w 1995 roku.
Internacjonalizacja (i18n) - jest procesem projektowania aplikacji, tak aby można je było dostosować do różnych języków i regionów (na całym świecie), bez konieczności dokonywania zmian technicznych w aplikacji. Np. ciągi zawierające teksty aplikacji zawarte są w osobnym, nieskompilowanym pliku, który może być edytowany bez specjalistycznych narzędzi i wiedzy.

Lokalizacja (l10n) - to proces dostosowania zinternacjonalizowanego oprogramowania do danego regionu lub języka przez dodanie elementów przystosowanych dla konkretnego odbiorcy i przetłumaczonego tekstu.
Różnica między internacjonalizacją i lokalizacją, jest subtelna, ale istotna.
Internacjonalizacja to dostosowanie produktów do ewentualnego wykorzystania praktycznie wszędzie, a *lokalizacja* to przystosowanie do warunków lokalnych, przez co produkt zlokalizowany nadaje się do wykorzystania tylko w konkretnych lokalizacjach.
Internacjonalizacja odbywa się raz dla konkretnego produktu, a lokalizacja jest wykonywana raz dla każdej kombinacji językowej.
[Wiele osób, zmęczonych pisaniem tych długich słów w kółko, wzięło zwyczaj pisania zamiast nich **i18n** i **l10n**. Czyli ze względu na długość słowa skracane do: i18n - liczba 18 oznacza ilość znaków między pierwszym „i" i ostatnim „n" w słowie internationalization, oraz odpowiednio L10n. (Kapitalizacja litery „L" w L10n pozwala odróżnić ją od małej litery „i" w i18n.) **]**

http://www.gnu.org/software/gettext/manual/gettext.html#Concepts

PoEdit korzysta z bibliotek gettext.
Idea **`gettext`** jest prosta, zamiast ręcznie wpisywać tekst w kodzie, np.:
`print("To jest jakiś tekst")`, wpisujemy **`print(_("To jest jakiś tekst"))`**
i funkcja makro **`_`** tłumaczy podany tekst zgodnie z podanymi przez Nas tłumaczeniami dla *wybranego* języka.

System gettext nastawiony jest do wykorzystania w dowolnym języku programowania, dlatego w jego skład wchodzi program xgettext, zdolny do utworzenia szablonu do tłumaczenia z źródła na wystarczająco dużej liczbie języków w tym - C, C++, C#, Perl, Python, Lisp ...
Ale tylko w przypadku, gdy autor wykorzysta ("**`import gettext`** oraz **`gettext.install()`**").

Wywołujemy funkcję gettext, aby szukała właściwego łańcucha w czasie wykonywania wtyczki przy użyciu oryginalnego angielskiego ciągu, jako klucza do słownika języka, który jest aktualnie aktywny.
Jeśli aktywny jest język angielski, *tłumaczenie nie jest wykonywane*, jeśli aktywny jest jakiś inny język np. ***pl***, tłumaczenie jest wykonywane, jeśli odpowiedni ciąg zostanie znaleziony.
Słowniki są to skompilowane pliki (pliki z rozszerzeniem **.mo** - oznacza to plik **Machine Object**).
Aby utworzyć pliki **.mo**, trzeba najpierw przeanalizować wszystkie kody źródłowe narzędziem gettext i wygenerować plik **.pot** (**.pot** oznacza **Portable Obiekt Template**), który jest po prostu sformatowanym plikiem tekstowym, który zawiera wszystkie angielskie ciągi (łańcuchy), które muszą być przetłumaczone.
Każdy łańcuch Angielski działa jako identyfikator wiadomości dla tego łańcucha. Poniżej każdego angielskiego ciągu jest miejsce dla przetłumaczonej wersji tego ciągu.
Kopiujemy plik **.pot** w **.po** (.**po** oznacza plik **Portable Object**) i przekazujemy go człowiekowi do tłumaczenia. Następnie musimy skompilować przetłumaczony plik **.po** do pliku **.mo** który umieszcza się w jednym z katalogów **LC_MESSAGE** Naszego systemu.
Katalogi te są nazwane naturalnymi języka, którego dotyczą. Kody obsługiwanych naturalnych języków podano w: http://www.gnu.org/software/gettext/manual/gettext.html#Language-Codes ISO 639 .
Jest to zbiór dwuznakowym kodów językowych.
Aplikacje zazwyczaj umieszczają różne pliki **.mo** wewnątrz docelowych konkretnych językowych podkatalogach w katalogach **./local**.
Poniższe drzewo katalogów pokazuje jak będzie wyglądać drzewo podkatalogów dla angielskiego (en), polskiego (pl) i francuskiego (es).

```
./locale/en/LC_MESSAGES
./locale/pl/LC_MESSAGES
./locale/fr/LC_MESSAGES
```

Powtórzmy, pliki **.mo** dla każdego języka aplikacji są przechowywane w katalogach kodów językowych wewnątrz podkatalogu **LC_MESSAGES**. Na przykład, plik słownika **.mo** znajduje się wewnątrz **./locale/pl/LC_MESSAGES**.
Ścieżką dostępu do pliku określamy drogę jaką należy przebyć z katalogu głównego do danego pliku, (poprzez strukturę katalogów). Jest ona ciągiem nazw podkatalogów oddzielonych znakiem "/" (slash) lub "\"(backslash) określanym mianem separatora.
Nazwy katalogów w ścieżkach oddzielamy w Linux znakiem "/", (odwrotnie niż w Windows).
`"."` oznacza katalog bieżący,
`".."` katalog nadrzędny
Podsumujmy:
Wtyczka musi zawierać następujący kod:

- musi importować moduł gettext - **`import gettext`**.
- musi wywołać **`gettext.install()`**, aby zainstalować funkcję **`_()`** wewnątrz słownika aplikacji.

To wywołanie określa domenę tłumaczeń,

1. ***Unikalna Nazwa*** *pliku tłumaczenia*

[*składa się z dwóch członów, pierwszy to* ***`gimp20`*****,** a drugi *odpowiada plikowi określonego słownika czyli* ***xyz.mo*** *- który zwykle jest nazwą wtyczki* ***xyz*** *(bez rozszerzenia* ***.mo****)*]

- poszukiwana przez funkcję ***`_()`*** *translatora gettext, zainstalowanego przez* ***`gettext.install()`***

2. położenie stosowanej **domeny tłumaczeń** `gimp.locale_directory` czyli podkatalogów **LC_MESSAGES**
3. oraz czy jest używane `unicode`.

Przykład wywołania słownika,

```
gettext.install('gimp20-xyz', 'gimp.locale_directory', unicode=True)
```

- Dla każdego obsługiwanego języka prezentacji, program musi stworzyć gettext.

Fragment przykładowej wtyczki:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-  # charset = zestaw znaków, aby gettext prawidłowo przetłumaczył w pliku znaki unicode.
from gimpfu import *
import os, sys, glob  # dodane przykładowo, zależą od potrzeb wtyczki
import locale         # dodane przykładowo, zależą od potrzeb wtyczki
import gettext # włącza biblioteki gettext do wtyczki. Spowoduje to możliwość utworzenia funkcji _() do tłumaczeń.
gettext.install("gimp20-xyz", gimp.locale_directory, unicode=True )
```

W Pythonie stosowane są trzy sposoby na tworzenie ciągów – łańcuchów znaków.

1. Ciąg w cudzysłowie, np. **`"Zbyma".`**
2. Ciąg w apostrofach, np. **`'Zbyma'`**.
3. Ciąg w potrójnym cudzysłowie (bądź potrójnych apostrofach): **`"""Zbyma"""`**.

Każdy z tych trzech rodzajów *stringów* (bo tak właśnie, z angielska wszyscy programiści mówią na cągi znaków) może zaczynać się od dodatkowego modyfikatora funkcji makro, czyli wywołanie funkcji **_ ( ).**
W literaturze anglojęzycznej stosowany jest zwrot "*owijanie*" stringów (ciągów) funkcją.

Dopóki ta funkcja nie jest określona program edytuje tylko oryginalne ciągi.

W wtyczkach spotkamy się również z funkcją:

```
N_() => # define N_(String)
```

która zaznacza ciągu do tłumaczenia, który zostanie zastąpiony *nieprzetłumaczonym* wyrażeniem w czasie wykonywania. Jest to przydatne w sytuacjach, gdy tłumaczone łańcuchy nie mogą być bezpośrednio stosowane.

Jeżeli wtyczka nie ma informacji o językach, to to czy jest zlokalizowana (przynajmniej częściowo) możemy sprawdzić w pliku tej wtyczki szukając w nich ciągów ze znakami _( ) jak podano będącymi nazwami funkcji lokalizujących. Jeżeli takie ciągi znaków znajdziemy oznacza to że możemy skorzystać z programu PoEdit.

Jeżeli takich oznaczeń ciągów nie ma to znaczy że wtyczka nie jest przygotowana do lokalizacji i jeżeli uprzemy się żeby z niej korzystać, powinniśmy ją do lokalizacji przygotować opakowując wszelkie występujące we wtyczce teksty w funkcję _("ciąg").

Wobec powyższego, jeśli autor wtyczki nie przygotował jej poprzez "owinięcie" ciągów funkcją, które wskazane jest przetłumaczyć. Jesteśmy jako tłumacze zmuszeni oznaczyć te ciągi funkcją np.:

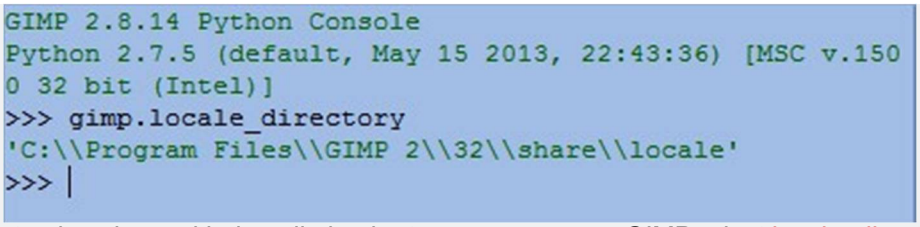
```
(PF_BRUSH,   'brush',       _("Brush"),
```

Wywołanie domeny dodawane jest do ***register*** wtyczki, w celu zlokalizowania tłumaczeń wtyczki.
Będzie to ***krotka***, która składa się z pliku tłumaczeń i ścieżki do katalogu, w którym zainstalowane są pliki tłumaczeń. Np.:

```
domain=("gimp20-xyz", gimp.locale_directory).
```

Gdzie: `gimp.locale_directory` powinna to być ścieżka bezwzględna, czyli:

C:\Program Files\GIMP 2\share\locale



Zrzut pokazuje przykład co dla konkretnego używanego GIMP-a jest lacale_directory
Ponieważ używam GIMP-a Windows, korzystam w ścieżce z separatora - odwrotnego ukośnika **"\"** (*backslash*), dlatego GIMP doda dla każdego jego wystąpienia oznaczenie **ucieczki** (**escape)**, ponieważ (backslash) **"\"** to znak specjalny w Pythonie.
Dokładnie to samo zobaczymy dla określonej wtyczki przeglądając plik **pluginrc**:

```
"C:\\Program Files\\GIMP 2\\32\\share\\locale"
```

Uwaga:
Python pozwala używać stylu ukośnika OS X / Linux (**slashes**) "**/**", nawet w systemie Windows.

Czyli, każda wtyczka do tłumaczenia **musi** korzystać z unikalnej nazwy pliku tłumaczeń oraz ***adresu domeny***, pod którą można umieścić plik. Przykładowy dalszy identyfikator

\pl\LC_MEESAGE

który określa, że pliki językowe (**xyz**) np. pl, są w podkatalogu LC_MESSAGES (zmienna środowiskowa).
Każdy LC_MESSAGES zawiera zestaw tłumaczeń dla różnych wtyczek na dany język.

Podsumowanie, co to jest plik *.pot .po* i *.mo*?

Jak już podano, GIMP używa rozszerzeń plików **.pot, .po i .mo** dla plików tłumaczeń.
Wykorzystują one GNU Gettext format.

Co oznacza skrót **POT**

Rozszerzenie pliku **.pot** oznacza Portable Object Template **Szablon obiektu** Portable
Jest to plik, który pojawia się podczas wyodrębniania tekstów z aplikacji. . Plik ten zawiera tylko oryginalne teksty, które wymagają przetłumaczenia. Normalnie można wysłać ten plik do tłumaczy.
Są plikami szablonów dla plików PO
(Uwaga: Microsoft również wykorzystuje POT jako rozszerzenie dla plików szablonów programu PowerPoint, ale to nie to samo).
Pliki **.pot** będą miały wszystkie ciągi do tłumaczenia (części **msgstr**) pozostawione puste, na przykład:

```
msgid "Hello world"
msgstr ""
```

Co oznacza skrót **PO**

Pliki z rozszerzeniem **.po** (Portable Object) są to pliki, które zawierają rzeczywiste tłumaczenia. Każdy język ma swój własny plik **.po**, na przykład, dla polskiego byłby **pl_PL.po**, dla amerykańskiego angielskiego może być **en_US.po**.
Źródło każdego pliku **.po** jest bardzo proste; są to po prostu pary tekstu
- ***msgid*** zawiera faktyczny tekst w kodzie i
- ***msgstr*** zawiera tekst, który jest tłumaczeniem
Na przykład, we francuskim pliku **.po** możemy mieć tłumaczenie **str** (*ciągu, napisu*) w następujący sposób:

```
msgid "Hello world"  #"Witaj świecie"
msgstr "Bonjour le monde"
```

Jeśli nie ma podanego tłumaczenia w **msgstr**, wyświetlony zostanie domyślny ciąg **msgid**.

Ponieważ pliki **.po** są po prostu plikami tekstowymi, tym samym mogą być edytowane za pomocą dowolnego edytora tekstu, ale istnieje również wiele dostępnych narzędzi do ich łatwiejszej edycji.
Program GIMP może zawierać wiele plików **.po** dla jednego języka (ale dla różnych wtyczek).

Co oznacza skrót **MO ?**

Plik - **.mo** (Machine object), jest zoptymalizowaną skompilowaną wersją maszynową wykonalnego pliku **.po**. [*language.po*], instalowanym w katalogu locale systemu.
Nie jest on czytelny dla człowieka, więc raczej nie próbujemy edytować go ręcznie. Zawsze generujemy pliki **.mo** używając **PoEdit** lub jakiegokolwiek innego narzędzia do przetłumaczenia.

Co to jest PoEdit?

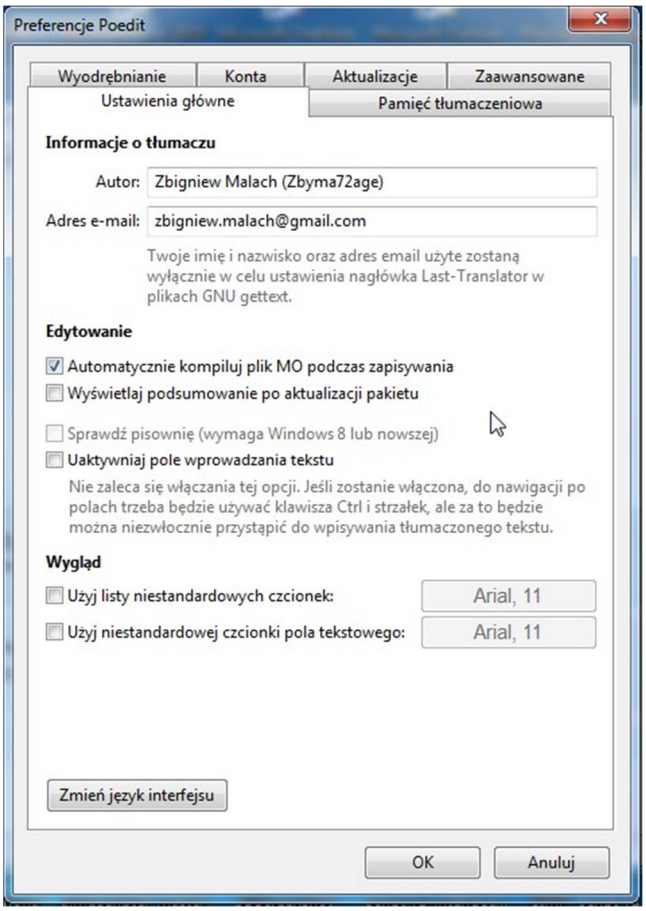
PoEdit http://poedit.net/ to program, który oferuje najlepszą drogę do tłumaczenia wtyczek GIMP-a (używa gettext). https://poeditor.com/
Dzięki specyficznemu podejściu, bez rozpraszania uwagi, będziemy z nim tłumaczyć szybko i łatwo.
Pobieramy go bezpłatnie http://poedit.net/download .
Istnieją wersje dla systemów Windows, Mac OS i Linux.
Obsługiwane formaty:

oraz wiele innych (https://poeditor.com/help/#SupportedFormats).

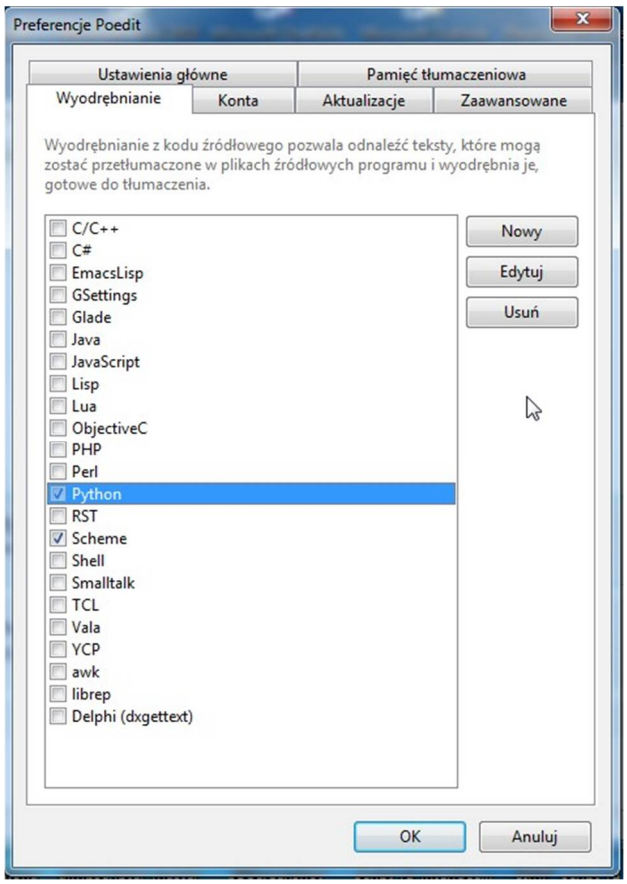
[Niektórzy używają narzędzia lokalizacji dla Pythona: https://poeditor.com.
Podobno ma bardzo łatwy w użyciu interfejs użytkownika, a niektórzy ludzie opracowali również scenariusze specjalnie dla Pythona, aby pobieranie plików było łatwiejsze.]

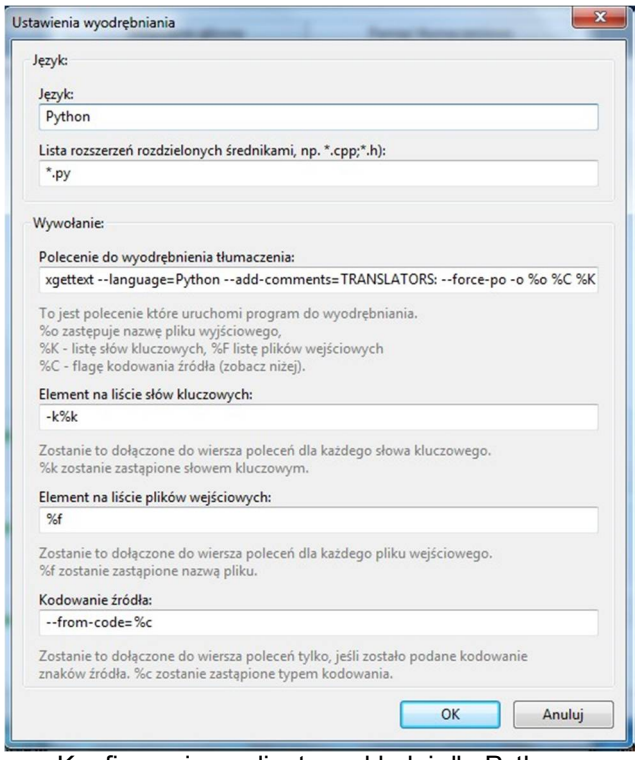
1. Po pierwsze, upewniamy się, że PoEdit został zainstalowany.
2. Upewniamy się, że mamy odpowiednie ustawienia: chcemy aby PoEdit był w stanie odczytać odpowiednie pliki

- Aby to zrobić, otwieramy PoEdit, a następnie przechodzimy do menu **Plik => Ustawienia…**. Otworzy się okno:



- Wprowadzamy dane na 3 pierwszych zakładkach
- Otwieramy zakładkę "Wyodrębnianie", a następnie na wyświetlonej liście zaznaczamy "**Python**" , następnie klikamy przycisk "Edytuj". Otworzy się kolejny okno:





Konfiguracja analizatora składni dla Pythona

Sprawdzamy i konfigurujemy ustawienia PoEdit dla Pythona:

- Język: Python
- Sprawdzamy czy w polu tekstowym pod tytułem "Lista rozszerzeń rozdzielonych średnikami (**np. *.cpp; *h**)", jeśli już tam nie jest, należy dodać rozszerzenie "***.py**", wpisując je w polu tekstowym.
- Wywołanie analizatora składni:
  polecenie uruchamiające xgettext oraz %o %K %C i inne (szczegóły opisano w oknie)
- Teraz dwa razy klikamy **OK**

10 kroków, aby wygenerować plik .pot korzystając z .py w PoEdit

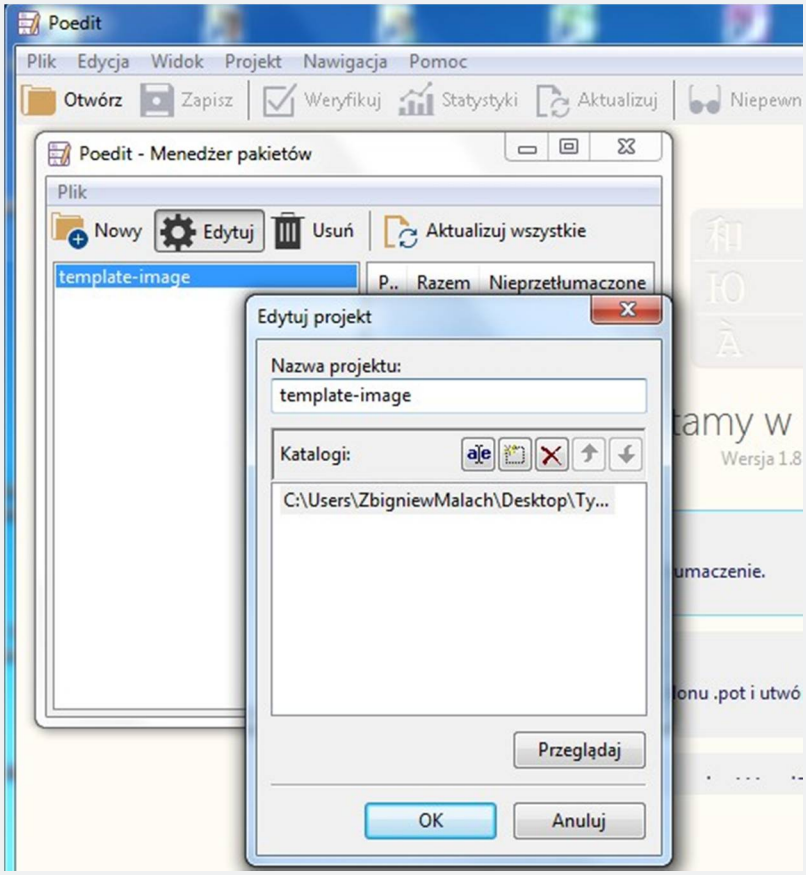
Krok 1

Pobieramy i instalujemy **PoEdit**. (Moja wersja stosowana w poradniku to ostatnia **PoEdit 1.8.4**)

Uwaga ze strony PoEdit:
Począwszy od wersji **PoEdit 1.8**, można edytować pliki POT, aktualizować je ze źródeł lub tworzyć nowe tłumaczenia.

Krok 2

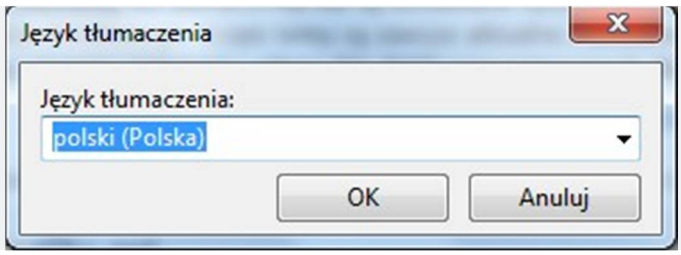
- Uruchamiamy PoEdit
- **Plik => Menedżer pakietów**
- Tutaj klikamy **Nowy** - tworzenie nowego projektu tłumaczenia
- Klikamy **Edytuj** i wpisujemy **Nazwa projektu** - umieszczamy nazwę swojej wtyczki
- Klikamy **Przeglądaj** - wybieramy ścieżkę do Naszego folderu z wtyczką **.py**, (*najprościej i najbezpieczniej, umieścić go np. na **Pulpicie***) i klikamy **OK**
- **Zamykamy** Menedżera pakietów



Proszę wziąć pod uwagę, że wszystkie wprowadzone dane są tylko dla tego przykładu.
Każdy musi umieścić własne.
Wprowadzone dane, w rzeczywistości nie są dowolne, ale są to kroki aby następnie utworzyć plik **.pot** dla stosowanego w poradniku "template-image"

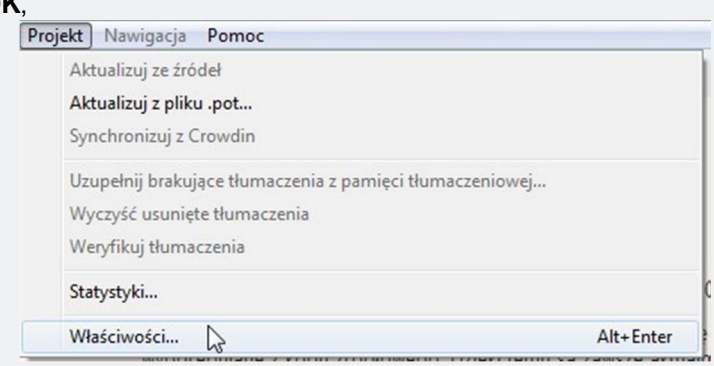
Krok 3

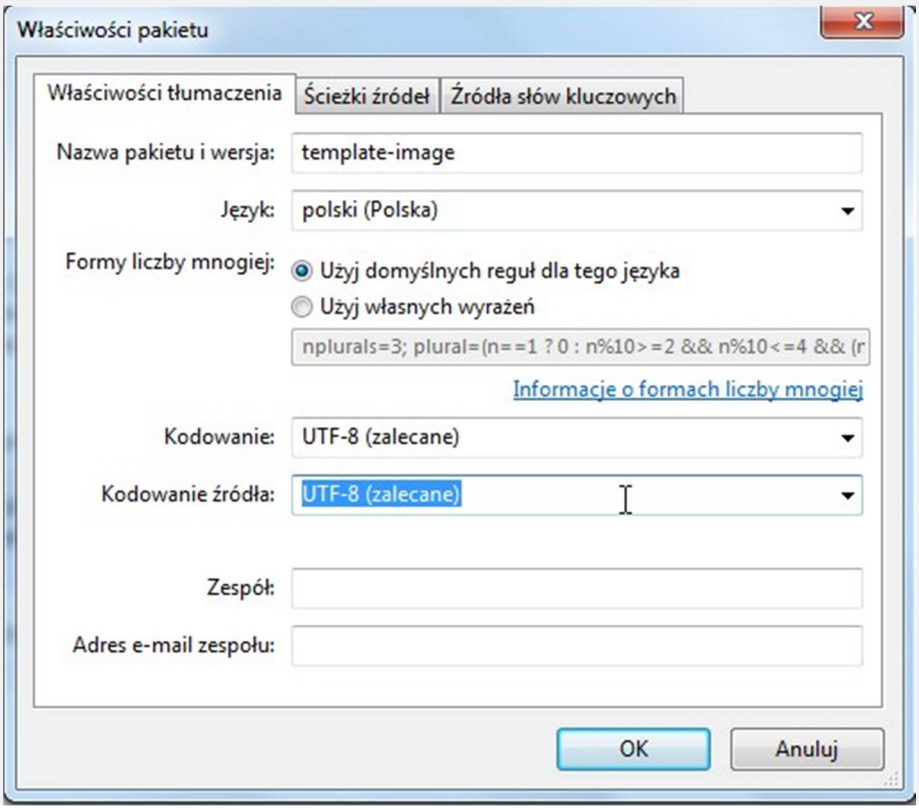
- **Plik => Nowe…** (Poedit zapyta na jaki język chcemy przetłumaczyć ciągi.)
- Wybieramy domyślny język tłumaczenia, **polski (Polska)**
- Klikamy przycisk **OK**,



Krok 4

- **Projekt => Właściwości**
- **"Nazwa projektu i wersja"** – najlepiej użyć tutaj nazwę wtyczki. Nie mamy tutaj na myśli nazwy folderu, ale aktualną nazwę wtyczki. Na przykład nazwa opracowywanej wtyczki jest "template image"
- Wybieramy Język: na jaki wtyczkę chcemy przetłumaczyć
- Kod źródłowy *charset* - wybieramy "UTF-8 (zalecane!)" (**WAŻNE: domyślnie generowany plik** .po deklaruje **charset** (*zestaw znaków*) jako ASCII. Oznacza to, że tłumaczenie ze znaków spoza ASCII sprowokuje błąd w programie. Aby rozwiązać ten problem, należy np. dla **pl** ustawić zawsze kodowanie znaków utf-8).
- Klikamy przycisk **OK**,





Krok 5

Teraz

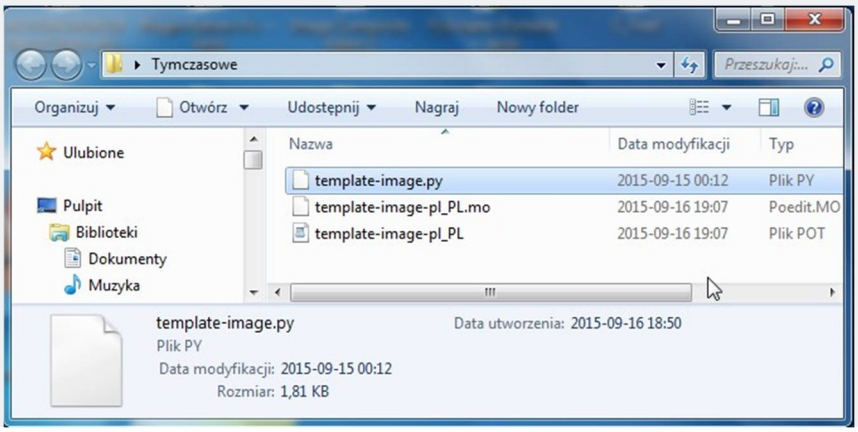
- **Plik => Zapisz**
- Nazwa pliku – moja rada jest taka stosować nazwę wtyczki i kod języka, tak więc wpisujemy:

template-image-pl_PL.pot, gdzie **pl_PL** jest kodem języka polskiego i Kraju Polska.
Oczywiście można np. wpisać "**pl_PL.pot**". Rozszerzenia pliku **.pot** nie znajdziemy na liście. Zamiast tego znajdziemy tylko **.po**. Nie przejmujemy się tym, wystarczy że sami wpiszemy rozszerzenie jako np. "**pl_PL.pot**". (Plik .pot jest w zasadzie listą wszystkich linii tekstu stosowanych w pliku wtyczki w języku angielskim.)

- Sprawdzamy katalog, gdzie mamy zamiar zapisać plik. Powinien to być Nasz utworzony tymczasowy folder wtyczki na - *Pulpit.*
- Klikamy przycisk **Zapisz**

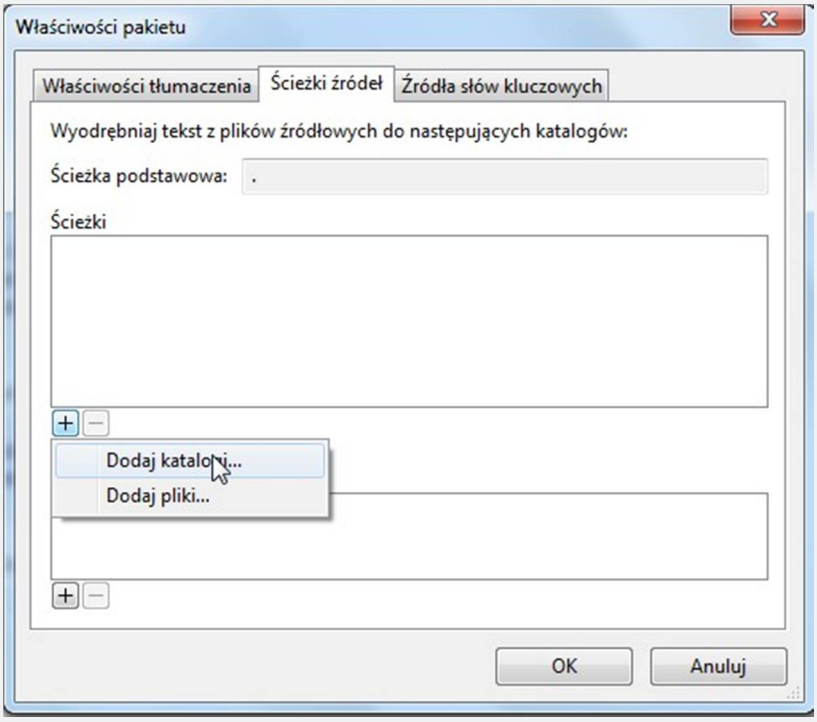
Krok 6

- Sprawdzamy folder Naszej wtyczki. Powinny tu pojawić się utworzone dwa nowe pliki: " **template-image-pl_PL.pot** " i " **template-image-pl_PL.mo**". (mam ustawione, aby PoEdit zawsze skompilował plik **.mo** podczas zapisywania zmian. Możemy to ustawić: **Plik => Preferencje** i na Zakładce **Edytor** wstawiamy (lub odznaczamy) zaznaczenie przy "**Automatycznie kompiluj plik .mo podczas zapisu"**.)
- Plik " **template-image-pl_PL.mo**" - usuwamy.



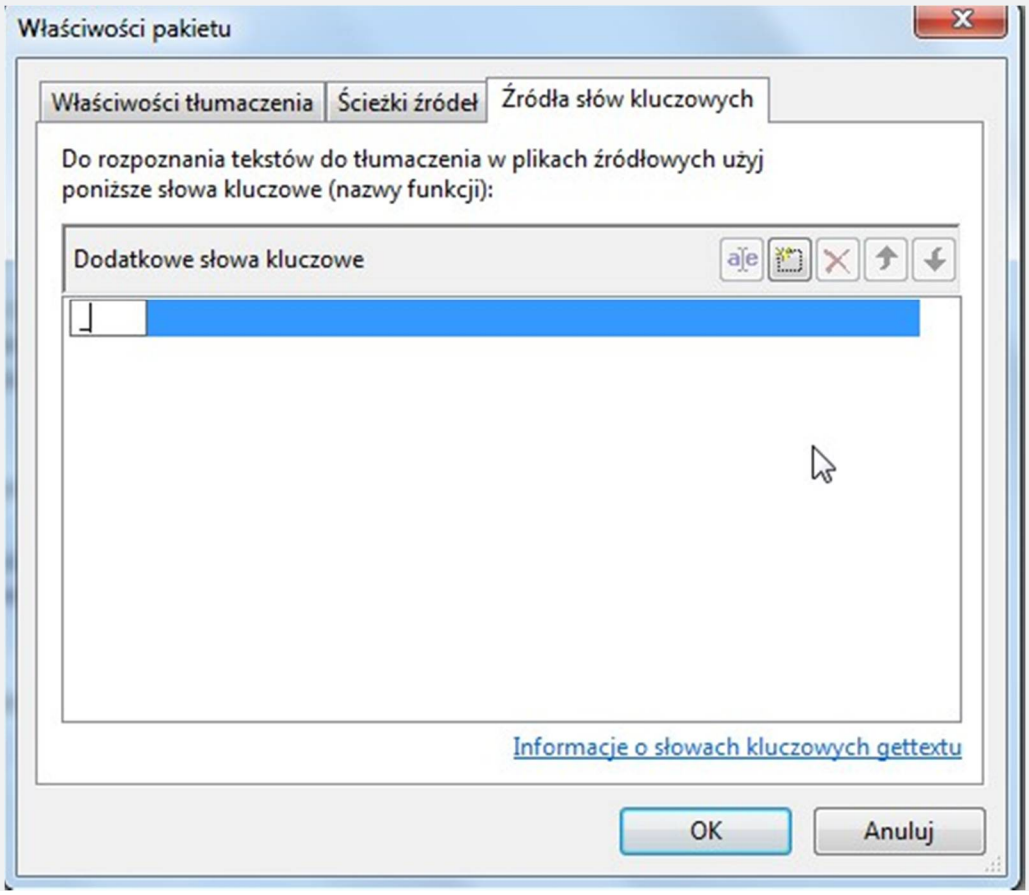
Krok 7

- **Projekt => Właściwości**
- Idziemy do zakładki "**Ścieżki źródeł**"
- W sekcji "Ścieżki" klikamy na **+** i "**Dodaj katalogi**"
- Teraz wystarczy umieścić kropkę **"."** w podświetlonej pozycji i nacisnąć klawisz OK, aby zostało zapisane (przez to PoEdit wie, że ma właśnie w tej ścieżce skanować pliki)



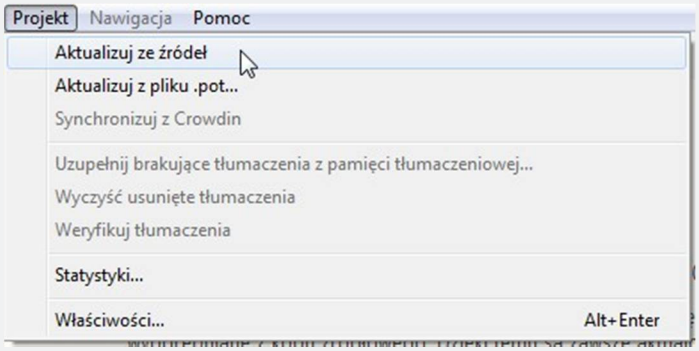
Krok 8

- Przechodzimy do zakładki "**Źródła słów kluczowych**"
- Klikamy na "Nowa pozycja" (przycisk 2, wygląda jak mały kwadrat)
- Wpisujemy słowo kluczowe _ (czyli funkcję gettext) , bo tylko ono jest używane we wtyczce .
- Klikamy przycisk **OK**,

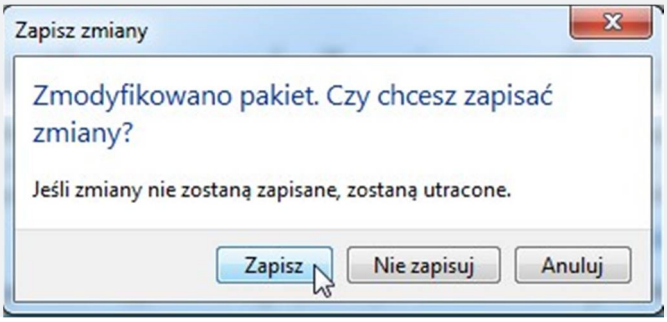


Krok 9

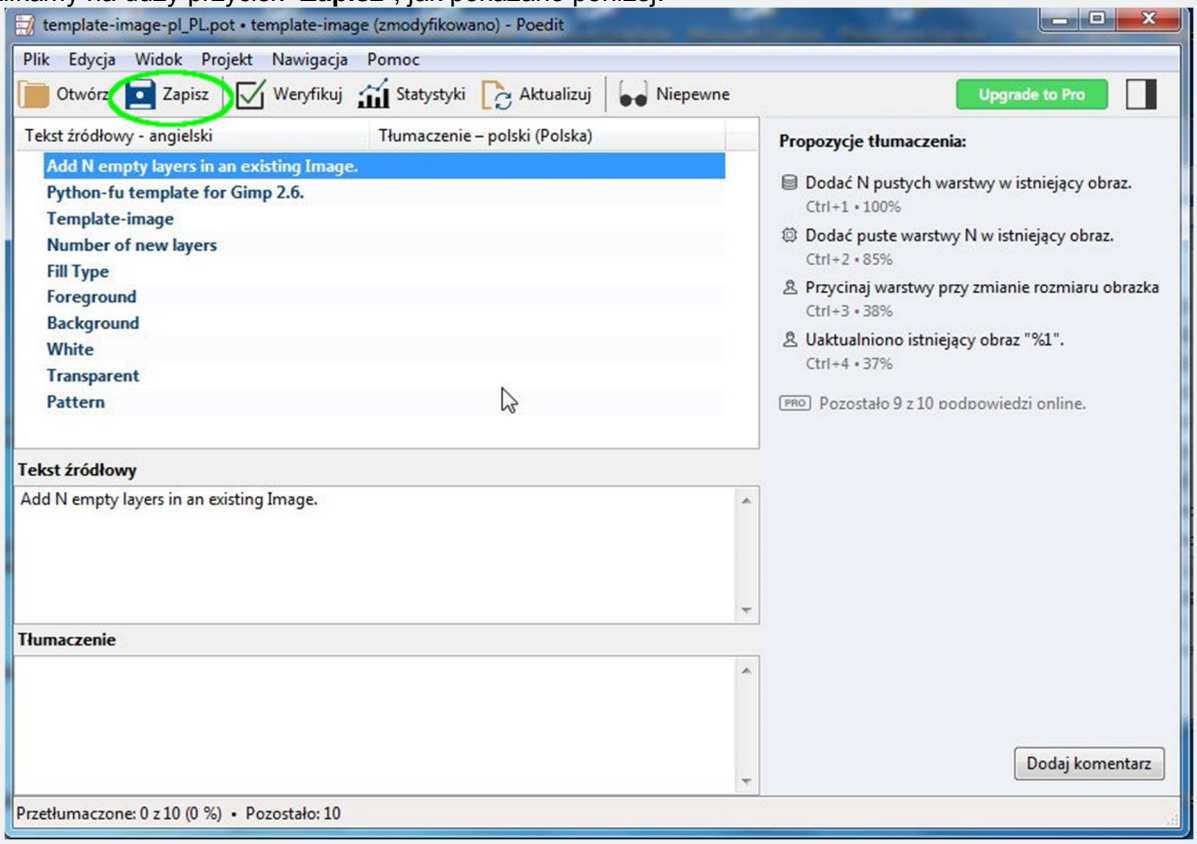
- **Projekt => "Aktualizuj ze źródeł"**



- Jeśli pojawi się okno z pytaniem o zapisanie zmian - klikamy **Zapisz**



- Teraz na ekranie zobaczymy, że plik **.pot** jest wypełniony wszystkimi danymi (frazami), które muszą być przetłumaczone.
- Klikamy na duży przycisk "**Zapisz**", jak pokazano poniżej:



Krok 10

- **Zamykamy program PoEdit**
  - W Naszym tymczasowym folderze wtyczki zobaczymy, że pojawił się plik: " **template-image-pl_PL.pot**

a także plik " **template-image-pl_PL.mo**" który został ponownie utworzony. ***Ponownie go usuwamy***.
Możemy sprawdzić zawartość pliku **.pot**

```
msgid ""
msgstr ""
"Project-Id-Version: template-image\n"
"POT-Creation-Date: 2015-09-18 17:02+0200\n"
"PO-Revision-Date: 2015-09-18 17:03+0200\n"
"Last-Translator: Zbigniew Malach (Zbyma72age)
<>>>>>>>>>>>>>>>>>.com>\n"
"Language-Team: \n"
"Language: pl_PL\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"X-Generator: Poedit 1.8.4\n"
"X-Poedit-Basepath: .\n"
"Plural-Forms: nplurals=3; plural=(n==1 ? 0 : n%10>=2 && n%10<=4 &&
(n%100<10 "
"|| n%100>=20) ? 1 : 2);\n"
"X-Poedit-SourceCharset: UTF-8\n"
"X-Poedit-KeywordsList: _\n"
"X-Poedit-SearchPath-0: .\n"

#: template-image.py:29
```

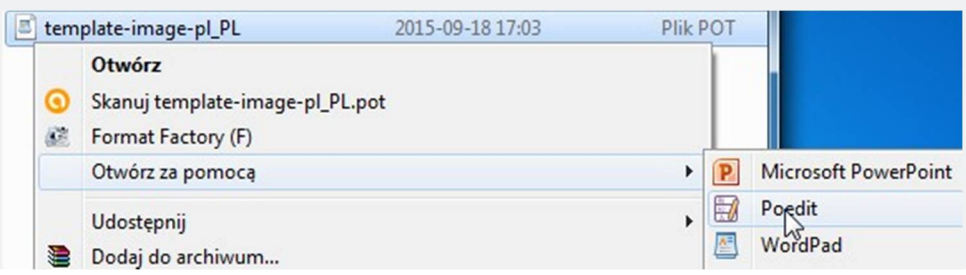
```
msgid "Add N empty layers in an existing Image."
msgstr ""

#: template-image.py:30
msgid "Python-fu template for Gimp 2.6."
msgstr ""

#: template-image.py:48
msgid "Template-image"
msgstr ""

#: template-image.py:53
msgid "Number of new layers"
msgstr ""

#: template-image.py:54
msgid "Fill Type"
msgstr ""

#: template-image.py:55
msgid "Foreground"
msgstr ""

#: template-image.py:56
msgid "Background"
msgstr ""

#: template-image.py:57
msgid "White"
msgstr ""

#: template-image.py:58
msgid "Transparent"
msgstr ""

#: template-image.py:59
msgid "Pattern"
msgstr ""
```

Jak widać plik **.pot** ma określoną strukturę.
Uwaga: Linie z znakiem **#** nie są istotne. To wskazuje tylko numery linii w skrypcie, gdzie ciąg został znaleziony.

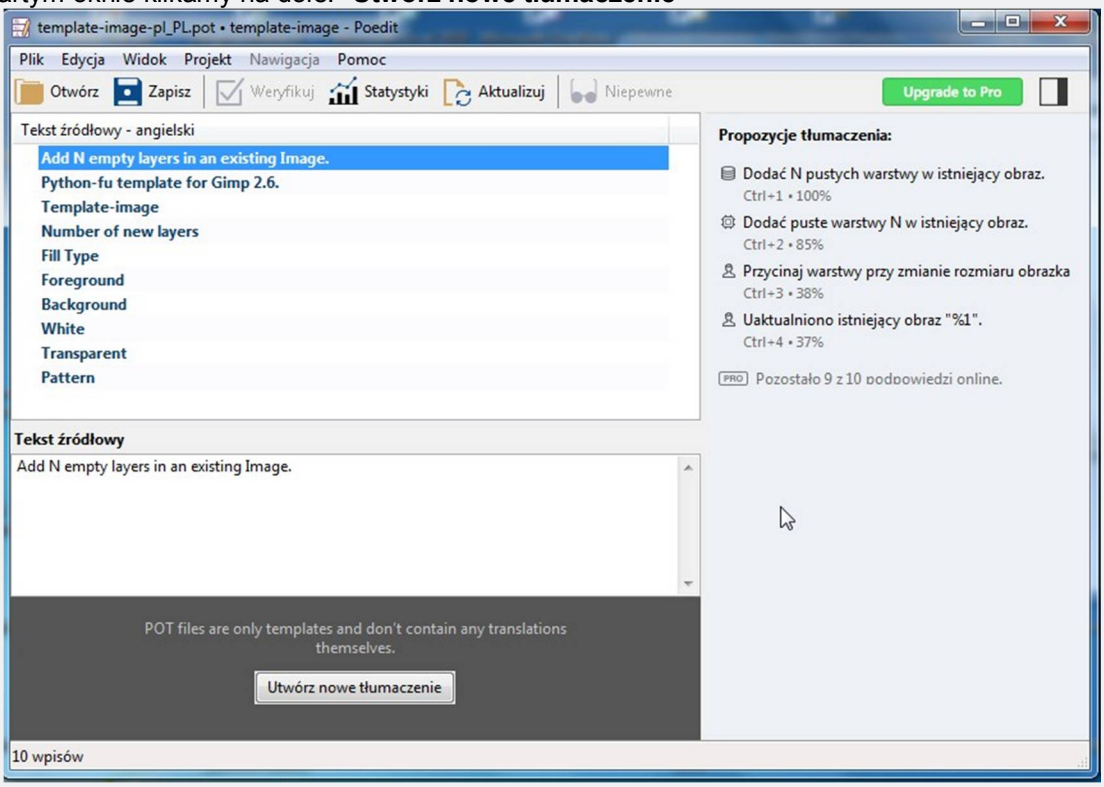
Składnia pliku jest całkiem przejrzysta. Komentarzy i praw autorskich nie tłumaczymy, parę z oryginalnych wierszy ciągów. Z pliku usunięte są wszystkie niepotrzebne ciągi z wyjątkiem linii ciągów niezbędnych dla tłumaczenia, wcześniej wskazano kodowanie UTF-8.

Od teraz dalej wszystko jest bardzo proste:

Uruchamiamy program PoEdit ponownie **lub**,
Otwieramy w PoEdit nasz plik **template-image-pl_PL.pot** metodą:



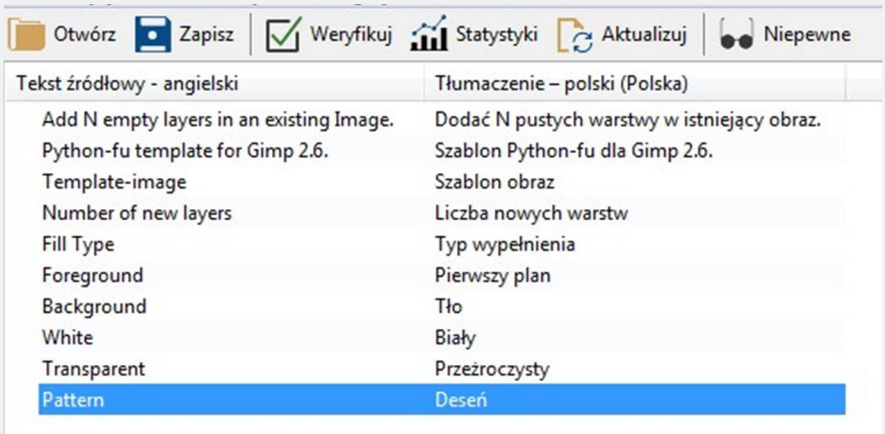
i w otwartym oknie klikamy na dole: "**Utwórz nowe tłumaczenie**"



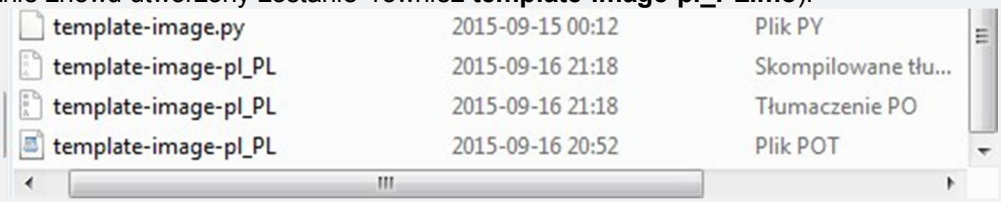
wykonujemy tłumaczenia fraz zaznaczonych we wtyczce jako funkcja **_( ),** które pojawiły **się** w oknie do przetłumaczenia. Warto zawsze sprawdzić czy tłumaczenia są poprawne, czy poprawnie zinterpretowaliśmy znaczenie fraz (Przykładowo: "name" dla obiektu martwego to nie "*imię*" tylko "*tytuł*" lub "*nazwa*".).
Ważne jest również, aby w tłumaczeniu były obecne wszystkie symbole typu `%1, %s` itd..
Tak samo znak przejścia do nowej linii `/n` powinien być stosowany co sensowną odległość (`/n` nie musimy umieszczać obowiązkowo w tym samym miejscu, resztę znaków "specjalnych" obowiązkowo).
[Tak przy okazji. Stosowany w PoEdit gettext skierowany jest wyłącznie do tłumaczenia wypełnionych gotowych propozycji i tłumaczenie propozycji niektórych słów i szablonów robi *niebezpieczne*... (na przykład gettext całkowicie nie obsługuje przypadków i rodzajów ale obsługuje rozróżnianie liczby pojedynczej i mnogiej. https://www.gnu.org/software/gettext/manual/html_node/Translating-plural-forms.html

```
msgid "%d file"
msgid_plural "%d files"
msgstr[0] "%d plik"
msgstr[1] "%d pliki"
msgstr[2] "%d plików" ]
```

Po prawej stronie okna pojawiają się propozycje tłumaczenia zaznaczonej frazy.



Po zakończeniu tłumaczenia
Plik => Zapisz jako: template-image-pl_PL.po
(automatycznie znowu utworzony zostanie również **template-image-pl_PL.mo**).

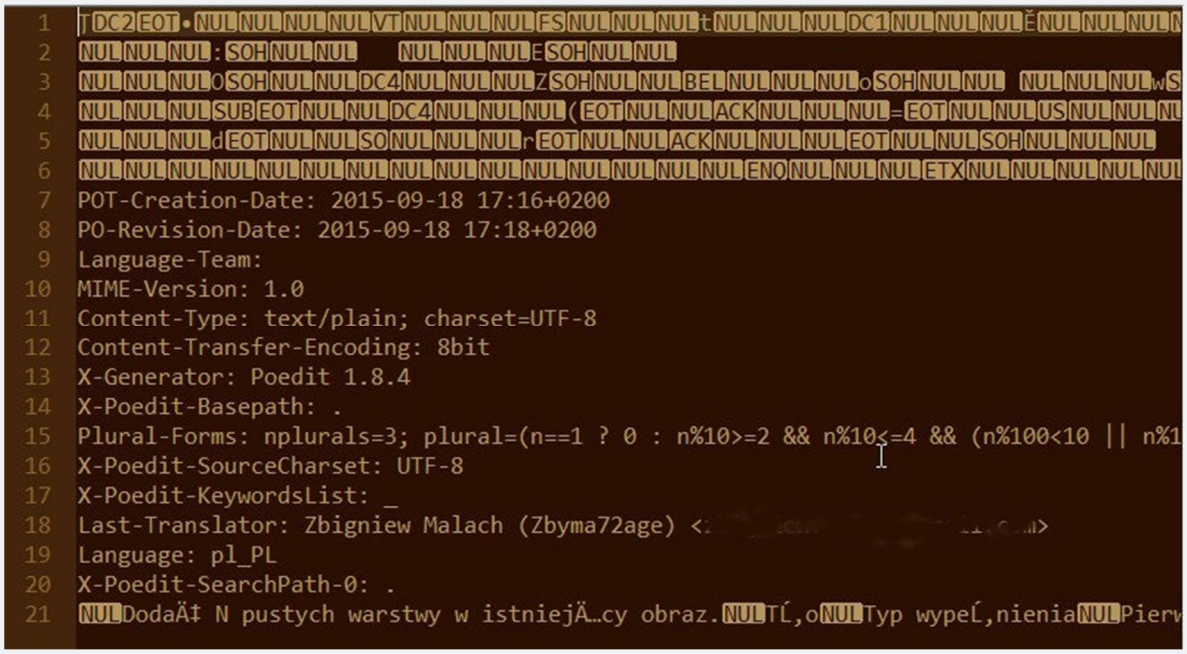


Uwaga:
Zwracam uwagę, że jeśli chcemy korzystać z skompilowanego pliku **template-image-pl_PL.mo** we wtyczce, musi zostać on przemianowany na: **gimp20-template** (**bez** mo).
Jak już podano zawartości pliku *.mo nie możemy edytować, ale można edytować plik ***.po**.
Poniżej podano poprawną edycję pliku w **Notepad++ :**

```
msgid ""
msgstr ""
"Project-Id-Version: template-image\n"
"POT-Creation-Date: 2015-09-18 17:16+0200\n"
"PO-Revision-Date: 2015-09-18 17:18+0200\n"
"Language-Team: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"X-Generator: Poedit 1.8.4\n"
"X-Poedit-Basepath: .\n"
"Plural-Forms: nplurals=3; plural=(n==1 ? 0 : n%10>=2 && n%10<=4 && (n%100<10 ||
n%100>=20) ? 1 : 2);\n"
"X-Poedit-SourceCharset: UTF-8\n"
"X-Poedit-KeywordsList: _\n"
"Last-Translator: Zbigniew Malach (Zbyma72age) <>>>>>>>>>>>>>>>>>>>>>>>.com>\n"
"Language: pl_PL\n"
"X-Poedit-SearchPath-0: .\n"

#: template-image.py:29
msgid "Add N empty layers in an existing Image."
msgstr "Dodać N pustych warstwy w istniejący obraz."

#: template-image.py:30
msgid "Python-fu template for Gimp 2.6."
msgstr "Szablon Python-fu dla Gimp 2.6."

#: template-image.py:48
msgid "Template-image"
msgstr "Szablon obraz"

#: template-image.py:53
msgid "Number of new layers"
msgstr "Liczba nowych warstw"

#: template-image.py:54
msgid "Fill Type"
msgstr "Typ wypełnienia"

#: template-image.py:55
msgid "Foreground"
msgstr "Pierwszy plan"

#: template-image.py:56
msgid "Background"
msgstr "Tło"
```

```
#: template-image.py:57
msgid "White"
msgstr "Białe"
#: template-image.py:58
msgid "Transparent"
msgstr "Przeźroczysty"

#: template-image.py:59
msgid "Pattern"
msgstr "Deseń"
```
można również np. w WordPad, ale wtedy przykładowo :
```
#: template-image.py:59
msgid "Pattern"
msgstr "DeseĹ„"
```

Pliki **.po** można również edytować on line w: https://localise.biz/free/poeditor . Pliki **.mo** z natury jako Machine Object nie są czytelne dla człowieka i nie można ich edytować bezpośrednio.
Oto jak wygląda plik **.mo** (Machine Object) edytowany w Notepad++:



Teraz po zmianie nazwy obu Naszych plików na dokładny język i kod kraju, kopiujemy pliki do odpowiedniego folderu.
Jesteśmy gotowi, aby zainstalować tłumaczenie wtyczki w odpowiednim katalogu.
W GIMP-ie istnieje odpowiednia hierarchia katalogów:

C;\Program Files\GIMP 2\share\locale

a w nim podkatalogi dla różnych języków - ru, en, pl, itp.; a w nich - w katalogach LC_MESSAGES, umieszczamy skompilowane pliki (**.mo** o maksymalnie unikalnej nazwie, która nie kolidują z innymi programami).
W naszym szkoleniowym przypadku, nie tworzymy w GIMP podkatalogu locale, podfolderów **pl\LC_MESSAGES** , bo one już istnieją i w ostatnim umieścimy nasz plik **gimp20-template**
Nazwy skompilowanych plików **.mo** są bardzo ważne i muszą pasować do żądanego ustawienia regionalnego (locale) we wtyczce, w przeciwnym razie wtyczka nie będzie działać.
Uwaga:
Konwencja nazewnictwa jest oparta na kodzie języka (np. **pt** w język portugalski), a następnie kod kraju (np. **_BR** dla Brazylii). Więc, portugalski w Brazylii będzie plik o nazwie **pt_BR.mo**.
Poniżej linki do pełnych list kodów języków i krajów, aby znaleźć dokładne ustawienia regionalne.

Patrz: **http://www.gnu.org/software/gettext/manual/html_chapter/gettext_16.html#Language-Codes**
http://www.gnu.org/software/gettext/manual/html_chapter/gettext_16.html#Country-Codes

Kilka dalszych wskazówek do tłumaczenia z zastosowaniem PoEdit i gettext:

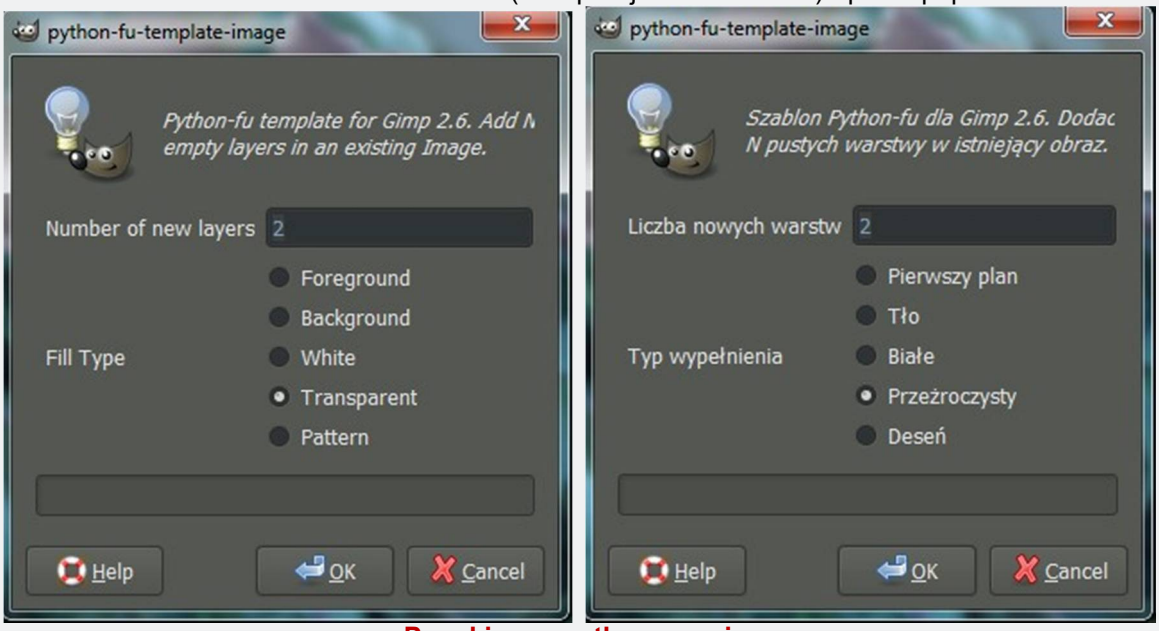
- W PoEdit, należy zachować szczególną ostrożność z ucieczką cudzysłowów ("), jeśli występują w źródle tłumaczenia (wszystkie znaki pomiędzy nimi są interpretowane jako dane tekstowe czyli napisy), przez dodanie poprzedzającego znaku ukośnika (\), przed każdym cudzysłowem (\").
  W przeciwnym razie tłumaczenie *może zostać obcięte*. PoEdit może dodać cudzysłów w naszej przetłumaczonej frazie, z wyjątkiem przypadków gdy znajdzie ucieczkę cudzysłowu w tłumaczeniu. Wystarczy sprawdzić i być ostrożnym z podwójnymi cudzysłowami.
- Najprawdopodobniej znajdziemy kilka rzeczy, takich jak znaki formatowania: **`%s;`** **`%d;`** lub **`%(modulo –`** początek definicji**`)`** w frazach do tłumaczenia. Są to symbole zastępcze dla niektórych danych np. (name, number), aby je wstawić w tym konkretnym miejscu. Na przykład, może się okazać, zdanie *"Witaj `%s`", gdzie %s zostanie zastąpione przez odpowiednią* ***nazwę*** *(name)w wtyczce.*

```
>>> #!/usr/bin/env python
>>> # -*- coding: utf-8 -*-
>>> # To wypisze "Zbyma ma 81 lat."
>>> name = "Zbyma"      # imie, Napis
>>> number = 81         # wiek, liczba całkowita dziesiętna
>>> print "%s ma %d lat." % (name, number)
Zbyma ma 81 lat.
>>>
```

  Należy przyjąć jako zasadę, aby obowiązkowo dołączać dowolny "symbol zastępczy" w swoim tłumaczeniu, kiedy go zobaczymy.

- Nie przekazujemy tłumaczenia w "Google Translate" do autora wtyczki. Zazwyczaj są to niepoprawne tłumaczenia.
- Przestrzegamy zasad interpunkcji w źródle i języku docelowym. Im więcej zaoferowanego dobrego tłumaczenia, tym więcej ludzi będzie go używać.
- Gramatyki różnych języków różnią się między sobą i te same słowa często należą do innych kategorii gramatycznych. Z tego względu jeżeli przekazujemy do funkcji ***_()*** frazy, których podmiot jest domyślny, czy po prostu pojedyncze słowa odwołujące się do kontekstu, musimy uważać aby nie popełnić błędu
- Sprawdź pisownię przed zapisaniem wtyczki!. Nie wysyłamy tłumaczenia pełnego błędnych wyrazów. Dlatego zawsze należy sprawdzić swoje tłumaczenie przed wysłaniem go do autora wtyczki.
- Do autora wtyczki wysyłamy zarówno pliki **.pot, .po** i **.mo** może on udostępnić nasze tłumaczenie innym osobom.

Sprawdzono w GIMP 2.8.6 Portable oraz 2.8.10 (kompilacja **32 bit** Partha) - praca poprawna.



Przed i po przetłumaczeniu.

Uwaga:

Jako wzoru użyłem wtyczki template-image z zestawu szablonów opracowanych przez: *Raymond Ostertag*, License - GNU/GPL ver. 3.1, 13 Styczeń 2009 **tested with Gimp-2.6.4**
http://gimpfr.org/contrib_template.php

Teraz przykład słynnej wtyczki `Hello World` *Akkana Peck* z 2010r:

Plik Oryginał:

```
#!/usr/bin/env python

# Hello World in GIMP Python

from gimpfu import *

def hello_world(initstr, font, size, color) :
    # First do a quick sanity check on the font
    if font == 'Comic Sans MS' :
        initstr = "Comic Sans? Are you sure?"

    # Make a new image. Size 10x10 for now -- we'll resize later.
    img = gimp.Image(10, 10, RGB)

    # Save the current foreground color:
    pdb.gimp_context_push()

    # Set the text color
    gimp.set_foreground(color)

    # Create a new text layer (-1 for the layer means create a new layer)
    layer = pdb.gimp_text_fontname(img, None, 0, 0, initstr, 10,
                                   True, size, PIXELS, font)

    # Resize the image to the size of the layer
    img.resize(layer.width, layer.height, 0, 0)

    # Background layer.
    # Can't add this first because we don't know the size of the text layer.
    background = gimp.Layer(img, "Background", layer.width, layer.height,
                            RGB_IMAGE, 100, NORMAL_MODE)
    background.fill(BACKGROUND_FILL)
    img.add_layer(background, 1)

    # Create a new image window
    gimp.Display(img)
    # Show the new image window
    gimp.displays_flush()

    # Restore the old foreground color:
    pdb.gimp_context_pop()

register(
    "python_fu_hello_world",
    "Hello world image",
    "Create a new image with your text string",
    "Akkana Peck",
    "Akkana Peck",
    "2010",
    "Hello world (Py)...",
    "",      # Create a new image, don't work on an existing one
    [
        (PF_STRING, "string", "Text string", 'Hello, world!'),
        (PF_FONT, "font", "Font face", "Sans"),
        (PF_SPINNER, "size", "Font size", 50, (1, 3000, 1)),
        (PF_COLOR, "color", "Text color", (1.0, 0.0, 0.0))
    ],
    [],
    hello_world, menu="<Image>/File/Create")

main()
```

Plik powyższej wtyczki **Hello World** *Akkana Peck*, poddamy operacji ***lokalizacji***.
W tym celu, plik wtyczki otwieramy w **Notepad++** i wprowadzamy lokalizację, czyli tłumaczenia na język polski ciągów, które pojawią się w interaktywnym oknie wtyczki:

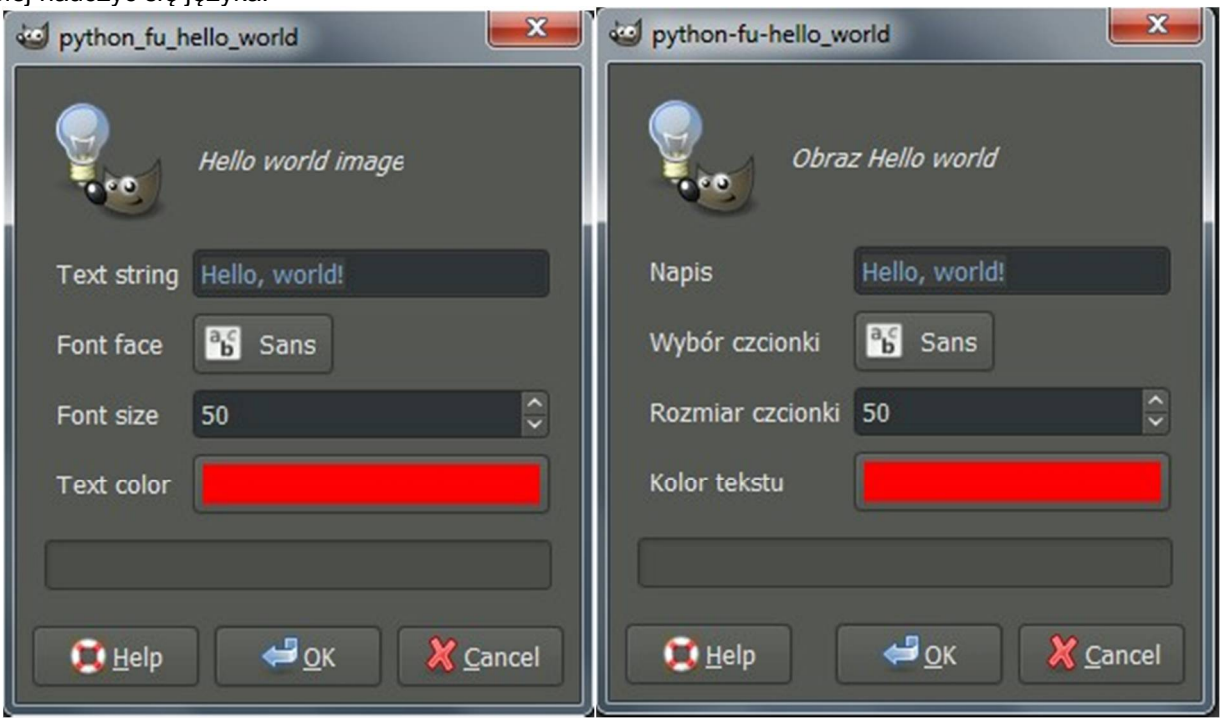
```
#!/usr/bin/env python
# -*- coding: utf-8 -*- # Dodano, deklaracja stosowanego kodowania znaków (pl)
# Hello World in GIMP Python
# http://www.efalk.org/Docs/Python/gimp-examples.html - stąd pobrano

from gimpfu import *  # Importujemy niezbędne moduły (odpowiednią bibliotekę),
# aby skrypt wiedział, z których wbudowanych funkcji ma korzystać.

def hello_world(initstr, font, size, color) :
    # Najpierw robimy szybki test poprawności czcionki
    if font == 'Comic Sans MS' :
        initstr = "Comic Sans? Are you sure?"

    # Tworzymy nowy obraz. Tymczasowo o rozmiarze 10x10 -- rozmiar będzie później
zmieniany.
    img = gimp.Image(10, 10, RGB)

    # Zapisujemy aktualny kolor pierwszoplanowy, aby móc z niego później
skorzystać:
    pdb.gimp_context_push()

    # Ustawiamy kolor czcionki (tekstu) na ten, który przekazujemy do funkcji
    gimp.set_foreground(color)

    # Teraz można przejść do tworzenia nowej warstwy, gdzie będzie się znajdował
tekst.
      # Pamiętamy cały czas o tym, że warstwa jest obiektem, więc trzeba ten
obiekt
      # przypisać do zmiennej, będzie - layer.
    layer = pdb.gimp_text_fontname(img, None, 0, 0, initstr, 10,
                                   True, size, PIXELS, font)

    # Musimy zmienić rozmiar obiektu, na którym warstwa została umieszczona,
      # aby były takie same (zmiana rozmiaru obrazu do rozmiaru warstwy).
      # Do tego możemy skorzystać z atrybutów warstwy, takich, jak: layer.width
oraz layer.height.
    img.resize(layer.width, layer.height, 0, 0)

    # Dodajemy warstwę Tło, które zawsze będzie przeźroczyste, jeżeli będziemy
pracowali na obiekcie Image
      # i nie nadamy mu tła). Dlatego zaraz dodamy kolor.
    # (Nie można było dodać tego najpierw, ponieważ nie znano, wielkość warstwy
tekstowej.)
    background = gimp.Layer(img, "Background", layer.width, layer.height,
                            RGB_IMAGE, 100, NORMAL_MODE)
    background.fill(BACKGROUND_FILL)
    img.add_layer(background, 1)

    # Jak na razie na ekranie nadal nic nie będzie, bo żaden obiekt nie został
wywołany.
      # Musimy zatem "utworzyć nowe okno obrazu na ekranie" wszystkie utworzone
obiekty,
      # czyli ten główny, na którym cały czas pracowaliśmy
    gimp.Display(img)
    # Aktualizujemy obraz na wyświetlaczu
    gimp.displays_flush()

    # Przywracamy "stary" kolor pierwszego planu:
    pdb.gimp_context_pop()
```

```
register(
    "hello_world",
    "Obraz Hello world ",
    "Tworzymy nowy obraz z Naszego łańcucha tekstu",
    "Akkana Peck, Modified: Zbyma72age",
    "Akkana Peck",
    "2010",
    "_Hello world (Py)...", # Etykieta, nazwa punktu w menu, za pomocą której,
wtyczka będzie uruchamiana,
            # zastosowanie znaku podkreślenia "_" przed pierwszym znakiem
powinno spowodować, że możemy używać
            # skrótu klawiaturowego, wygoda jeżeli w Katalogu/podkatalogu
znajduje się więcej wtyczek. Klikamy na katalog i potem skrót.
    "",      # Tworzenie nowego obrazu, nie działa na istniejącym
    [
        (PF_STRING, "string", "Napis", 'Hello, world!'),
        (PF_FONT, "font", "Wybór czcionki", "Sans"),
        (PF_SPINNER, "size", "Rozmiar czcionki", 50, (1, 3000, 1)),
        (PF_COLOR, "color", "Kolor tekstu", (1.0, 0.0, 0.0))
    ],
    [],
    hello_world, menu="<Image>/File/Create")

main()
```

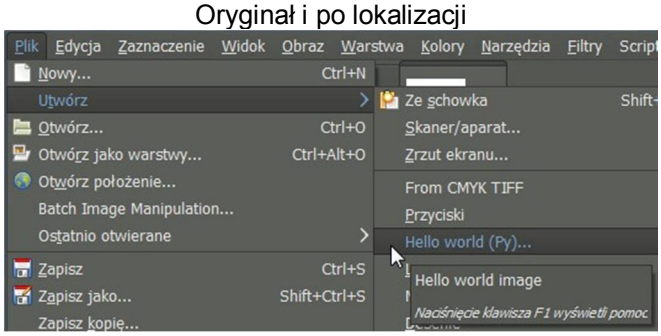
Po zakończeniu tłumaczenia poszczególnych ciągów, **Plik => Zapisz jako…,** możemy do oryginalnej nazwy wtyczki dodać jakieś oznaczenie np. **hello_world_lok.py** lub pozostawić nazwę nie zmienioną.

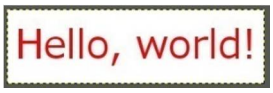
Uwaga:
Podane we wtyczce tłumaczenia wszystkich komentarzy, zostało przygotowane dla wykorzystania w innym poradniku. Kolorem czerwonym zaznaczono dodane ciągi komentarzy oraz przetłumaczone ciągi wyświetlane w oknie wtyczki.

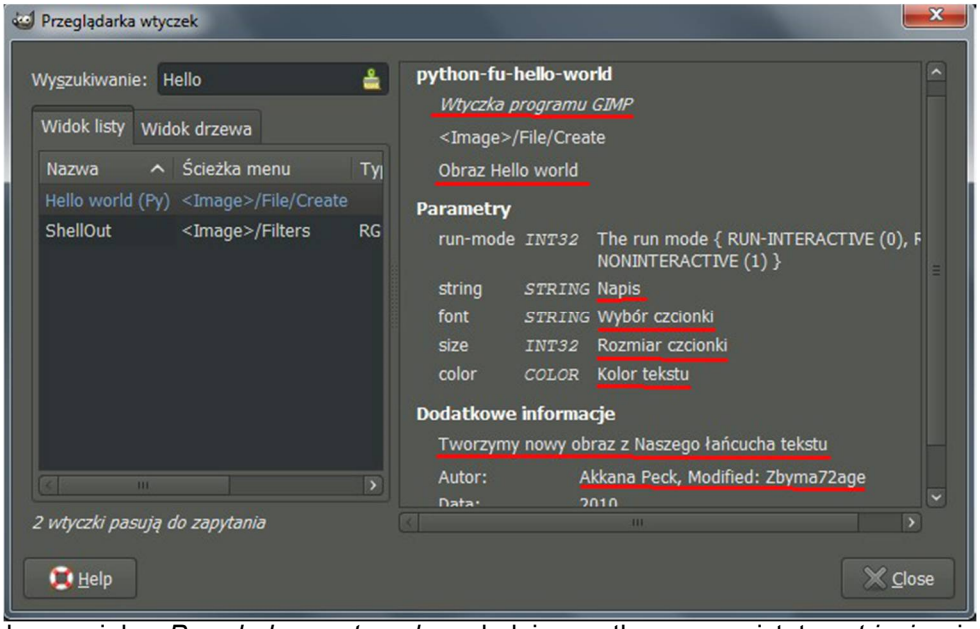
W nauce programowania, utrwaliła się taka tradycja, że za każdym razem, kiedy uczymy się nowego języka programowania, pierwszym programem jest "Hello World" — wszystko, co robi, to tylko wypisuje tekst "Hello World". Jak mówi Simon Cozens (autor świetnej książki pt. „Perl. Od podstaw"), jest to tradycyjne obrzędowe zaklęcie do bogów programowania [inkantacja (łac. incantare - oczarować)], by pozwolili Nam lepiej nauczyć się języka.



Oryginał i po lokalizacji







Pokazano jak w *Przeglądarce wtyczek* wyglądają przetłumaczone istotne *stringi* – ciągi.

Teraz - przygotowanie tej samej wtyczki `Hello World` *Akkana Peck* do Internacjonalizacji

Jak wspomniano powyżej, jeżeli wtyczka nie ma informacji o językach, to to czy jest zlokalizowana (przynajmniej częściowo) możemy sprawdzić w pliku tej wtyczki szukając w nich ciągów ze znakami **_( )** będącymi nazwami funkcji lokalizujących
Ponieważ takich oznaczeń ciągów nie ma to znaczy że wtyczka nie jest przygotowana do lokalizacji i możemy ją do Internacjonalizacji (*Umiędzynarodowienia*) przygotować, "owijając" w funkcję **_(**"ciąg"**)** ciągi we wtyczce, które wskazane jest przetłumaczyć bo pojawią się w interaktywnym oknie

W tym celu ponownie otwieramy oryginalny plik wtyczki w **Notepad++** i owijamy potrzebne ciągi:

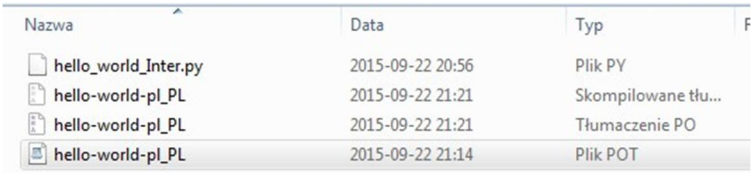
```
#!/usr/bin/env python
# -*- coding: utf-8 -*-  # charset = zestaw znaków,
# aby gettext prawidłowo przetłumaczył w pliku znaki unicode.
# Hello World in GIMP Python

from gimpfu import *
import gettext # włącza biblioteki gettext do wtyczki.
# Spowoduje to możliwość utworzenia funkcji _() do tłumaczeń.

gettext.install("gimp20-hello-world" , gimp.locale_directory, unicode=True)

def hello_world(initstr, font, size, color) :
    # First do a quick sanity check on the font
    if font == 'Comic Sans MS' :
        initstr = "Comic Sans? Are you sure?"
```

```
    # Make a new image. Size 10x10 for now -- we'll resize later.
    img = gimp.Image(10, 10, RGB)

    # Save the current foreground color:
    pdb.gimp_context_push()

    # Set the text color
    gimp.set_foreground(color)

    # Create a new text layer (-1 for the layer means create a new layer)
    layer = pdb.gimp_text_fontname(img, None, 0, 0, initstr, 10,
                                   True, size, PIXELS, font)
    # Resize the image to the size of the layer
    img.resize(layer.width, layer.height, 0, 0)

    # Background layer.
    # Can't add this first because we don't know the size of the text layer.
    background = gimp.Layer(img, "Background", layer.width, layer.height,
                            RGB_IMAGE, 100, NORMAL_MODE)
    background.fill(BACKGROUND_FILL)
    img.add_layer(background, 1)

    # Create a new image window
    gimp.Display(img)
    # Show the new image window
    gimp.displays_flush()

    # Restore the old foreground color:
    pdb.gimp_context_pop()

register(
    "hello_world",
    _("Hello world image"),
    _("Create a new image with your text string"),
    "Akkana Peck",
    "Akkana Peck",
    "2010",
    "_Hello world (Py)...",
    "",      # Create a new image, don't work on an existing one
    [
        (PF_STRING, "string", _("Text string"), 'Hello, world!'),
        (PF_FONT, "font", _("Font face"), "Sans"),
        (PF_SPINNER, "size", _("Font size"), 50, (1, 3000, 1)),
        (PF_COLOR, "color", _("Text color"), (1.0, 0.0, 0.0))
    ],
    [],
    hello_world, menu="<Image>/File/Create", # pamiętać o tym przecinku!!
    domain=("gimp20-hello-world", gimp.locale_directory)
      )

main()
```

Po zakończeniu "*owijania*" poszczególnych ciągów, **Plik => Zapisz jako…,** nazwę tego pliku wtyczki ustalamy przykładowo np. na: `hello-world_Inter.py`

Plik umieszczamy w tymczasowym folderze na **Pulpicie.**

Uruchamiamy PoEdit i
zgodnie z podaną metodologią na podstawie pliku wtyczki `hello_world_Inter.py` utworzono pliki:

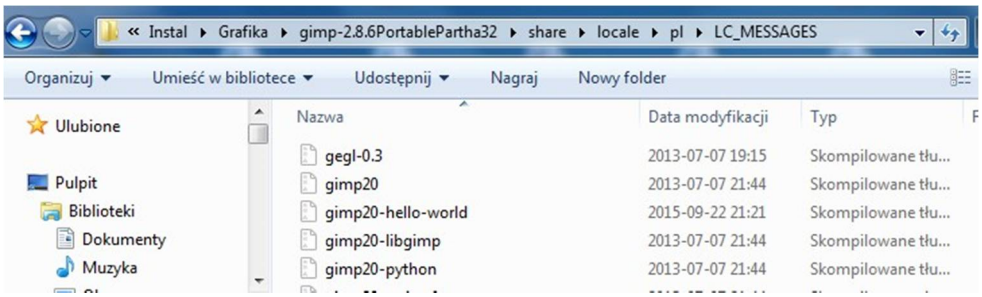


Zawartość pliku **`hello-world-pl_PL.po`** sprawdzamy np. w Notepad++:

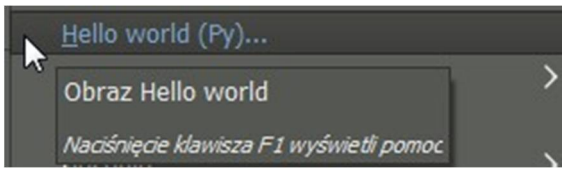
```
msgid ""
msgstr ""
"Project-Id-Version: Hello World\n"
"POT-Creation-Date: 2015-09-22 21:16+0200\n"
"PO-Revision-Date: 2015-09-22 21:21+0200\n"
"Language-Team: \n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
"X-Generator: Poedit 1.8.4\n"
"X-Poedit-Basepath: .\n"
"Plural-Forms: nplurals=3; plural=(n==1 ? 0 : n%10>=2 && n%10<=4 && (n%100<10 || n%
"X-Poedit-SourceCharset: UTF-8\n"
"X-Poedit-KeywordsList: _\n"
"Last-Translator: Zbigniew Malach (Zbyma72age) <                    >\n"
"Language: pl_PL\n"
"X-Poedit-SearchPath-0: .\n"

#: hello_world_Inter.py:53
msgid "Hello world image"
msgstr "Obraz Hello world"

#: hello_world_Inter.py:54
msgid "Create a new image with your text string"
msgstr "Tworzymy nowy obraz z Naszego łańcucha tekstu"

#: hello_world_Inter.py:61
msgid "Text string"
msgstr "Napis"
```

Zmieniamy nazwę skompilowanego pliku **`hello-world-pl_PL.mo`**, na **`gimp20-hello-world`**
i umieszczamy w:

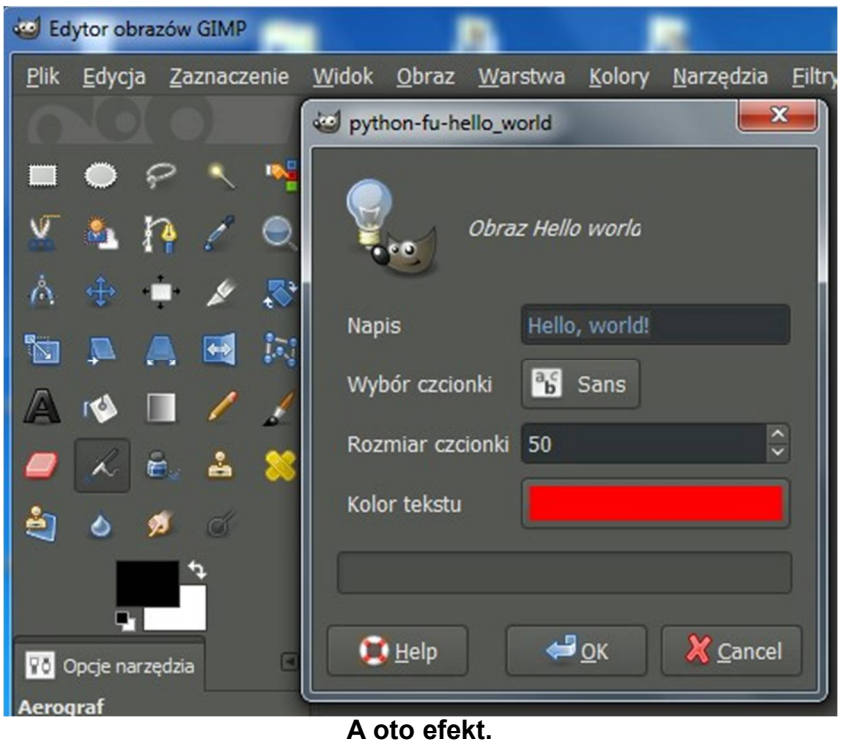
C;\Program Files\GIMP 2\share\locale\ pl\LC_MESSAGES



Jak widać zastosowano do testów **GIMP 2.8.6 Portable -kompilacji *Partha* - 32bit**



Proszę zwrócić uwagę na podkreślone **H**, ponieważ zastosowano **`"_Hello world (Py)..."`**
(powyżej w lokalizacji wtyczki umieszczono szczegółowe wytłumaczenie przeznaczenia kodu)



A oto efekt.

Pamiętamy także, aby po każdej aktualizacji wtyczki, zaktualizować także tłumaczenie.

Udowodnimy to poprzez równoczesne sprawdzenie, czy Nasze tłumaczenie jest tam gdzie powinno być umieszczone i czy działa poprawnie.

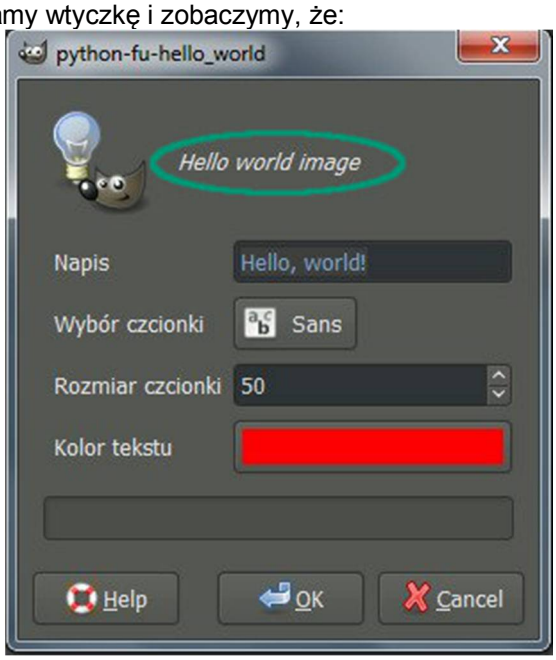
Plik tłumaczenia `gimp20-hello-world` w GIMP jest umieszczony tam gdzie poprzednio, nic nie zmieniono.

Teraz dla sprawdzenia metodą drag&drop *przeciągamy* z folderu GIMP-a **~\.gimp-2.8\plug-ins** plik wtyczki `hello_world_Inter.py` do okna **Notepad++** i przykładowo zmieniamy ciąg:
`_("Hello world image")` wprowadzając *dodatkową spację* `_("Hello world image "),`

```
register(
    "hello_world",
    _("Hello world image "),
```

zmianę zatwierdzamy w Notepad++

Plik => Zapisz.
Otwieramy GIMP-a, uruchamiamy wtyczkę i zobaczymy, że:



Biblioteka **gettext** nie może pobrać przetłumaczonego ciągu, z pliku tłumaczeń bo nie znalazła poprawnego tłumaczenia obecnego ciągu `_("Hello world image ")` w pliku wtyczki `hello_world_Inter.py`, ponieważ zmieniono ciąg w stosunku do tego, który znajduje się w pliku tłumaczeń `gimp20-hello-world`
W związku z tym ciąg został **zastąpiony** w oknie wtyczki *nieprzetłumaczonym* wyrażeniem.
W taki oto sposób pokazano, że wykonane tłumaczenie wtyczki było poprawne, oraz udowodniono, że po każdej aktualizacji (zmianie) w treści wtyczki musimy zaktualizować również tłumaczenie.

Wtyczka zmodyfikowana dla procedur obowiązujących w GIMP 2.8
(zaznaczono w których procedurach dokonano zmian)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-  # charset = zestaw znaków,
# aby gettext prawidłowo przetłumaczył w pliku znaki unicode.
# Hello World in GIMP Python

from gimpfu import *  # Importujemy niezbędne moduły (odpowiednią bibliotekę),
# aby skrypt wiedział, z których wbudowanych funkcji ma korzystać.
import gettext # włącza biblioteki gettext do wtyczki.
# Spowoduje to możliwość utworzenia funkcji _() do tłumaczeń.

gettext.install("gimp20-hello-world" , gimp.locale_directory, unicode=True)

def hello_world(initstr, font, size, color) :
    # Najpierw robimy szybki test poprawności czcionki
    if font == 'Comic Sans MS' :
        initstr = "Comic Sans? Are you sure?"

    # Tworzymy nowy obraz. Tymczasowo o rozmiarze 10x10 -- rozmiar będzie
później zmieniany.
    image = pdb.gimp_image_new(10, 10, RGB)

    # Zapisujemy aktualny kolor pierwszoplanowy, aby móc z niego potem
skorzystać:
    pdb.gimp_context_push()

    # Ustawiamy kolor czcionki (tekstu) na ten, który przekazujemy do procedury
    pdb.gimp_context_set_foreground(color)

    # Teraz można przejść do tworzenia nowej warstwy, na której będzie
umieszczony tekst.
      # Pamiętamy cały czas o tym, że warstwa jest obiektem, więc trzeba ten
obiekt
      # przypisać do zmiennej, będzie to - text_layer.
    text_layer = pdb.gimp_text_fontname(image, None, 0, 0, initstr, 10, True,
size, PIXELS, font)

      # Musimy zmienić rozmiar obiektu, na którym warstwa została umieszczona,
      # aby były takie same (zmiana rozmiaru obrazu do rozmiaru warstwy).
      # Do tego możemy skorzystać z atrybutów warstwy, takich, jak:
text_layer.weigth oraz text_layer.height.
    pdb.gimp_image_resize(image, text_layer.width, text_layer.height, 0, 0)

    # Dodajemy warstwę Tło, które zawsze będzie przeźroczyste, jeżeli będziemy
pracowali na obiekcie Image
      # i nie nadamy mu tła). Dlatego zaraz dodamy kolor.
    # (Nie można było dodać tego najpierw, ponieważ nie znano, wielkość warstwy
text_layer.)
    layer = pdb.gimp_layer_new(image, text_layer.width, text_layer.height,
RGB_IMAGE, "Background", 100, NORMAL_MODE)
    pdb.gimp_drawable_fill(layer, BACKGROUND_FILL)
    pdb.gimp_image_insert_layer(image, layer, None, 1)  # poprzednia procedura
img.add_layer()jest już przestarzała
```

```
    # Jak na razie na ekranie nadal nic nie będzie, bo żaden obiekt nie został
wywołany.
      # Musimy zatem "uworzyć nowe okno obrazu na ekranie" wszystkie utworzone
obiekty,
      # czyli ten główny, na którym cały czas pracowaliśmy
    display = pdb.gimp_display_new(image)
    # Aktualizujemy obraz na ekranie.
    pdb.gimp_displays_flush()

    # Przywracamy "stary" kolor pierwszego planu:
    pdb.gimp_context_pop()

register(
    "hello_world",
    _("Hello world image"),
    _("Create a new image with your text string"),
    "Akkana Peck",
    "Akkana Peck",
    "2010",
    "_Hello world (Py)...",  # Etykieta, nazwa punktu w menu, za pomocą której,
wtyczka będzie uruchamiana,
            # zastosowanie znaku podkreślenia "_" przed pierwszym znakiem
powinno spowodować, że możemy używać
            # skrótu klawiaturowego, wygoda jeżeli w Katalogu/podkatalogu
znajduje sie więcej wtyczek. Klikamy na katalog i potem skrót.
    "",      # Tworzenie nowego obrazu, nie działa na istniejącym
    [
        (PF_STRING, "string", _("Text string"), 'Hello, world!'),
        (PF_FONT, "font", _("Font face"), "Sans"),
        (PF_SPINNER, "size", _("Font size"), 50, (1, 3000, 1)),
        (PF_COLOR, "color", _("Text color"), (1.0, 0.0, 0.0))
    ],
    [],
    hello_world, menu="<Image>/File/Create", # pamiętać o tym przecinku dodając
domain!!
    domain=("gimp20-hello-world", gimp.locale_directory)
      )

main()
```

Tak zmodyfikowaną wtyczkę zapisano jako: `hello-world_Inter2.py`

Jak widać powyżej również działa poprawnie.

Sprawdzono również w niestabilnej wersji rozwojowej GIMP 2.9.1 64 bit Portable w kompilacji Partha.

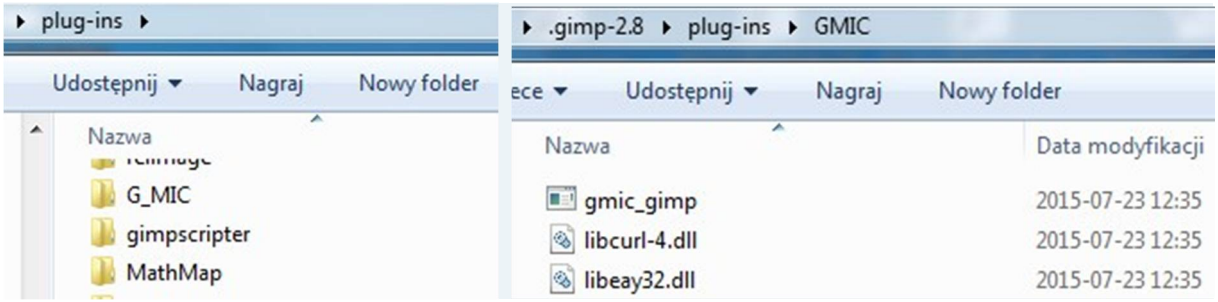
W finale muszą wspomnieć jeszcze, o kilku istotnych rzeczach:

Istnieje pewne zamieszanie wokół różnych 32 i 64 bitowych wersji systemu Windows, GIMP i Python.
* Jeśli używamy GIMP-a 64 bit, powinien być stosowany Python 64 bit oraz PyGtk 64 bit.
Co to oznacza ?.
32-bitowe wtyczki uruchomimy, jeśli dostarczamy wszystkie 32-bitowe biblioteki **DLL**, w tym wszystkie 32-bitowe biblioteki **DLL** w 64-bitowym GIMP-ie (rozszerzenia są oznaczane w Windows ***.dll***.)
DLL (z ang. *Dynamic-Link Library* - biblioteka łączona dynamicznie) – w środowisku Microsoft Windows biblioteka współdzielona (z ang. *shared library*), która przechowuje implementacje różnych podprogramów programu lub zasoby programu. Podprogramy i zasoby zawarte w bibliotece DLL mogą być wykorzystane bezpośrednio lub pośrednio (za pośrednictwem innej biblioteki DLL) przez dowolny plik wykonywalny, sama biblioteka DLL nie jest samodzielnym programem**!**.
Z różnych powodów zdarza się, że otrzymujemy komunikat o brakujących bądź uszkodzonych plikach dll, przyczyn takich problemów jest wiele począwszy od odinstalowania jakiejś wtyczki po problemy podczas uruchamiania, gdy brakuje jakiegoś składnika. Zazwyczaj komunikat brzmi następująco: Uruchomienie wtyczki nie powiodło się, nie odnaleziono określonej biblioteki ***.dll***.

Piekło DLL *('DLL hell')* *https://pl.wikipedia.org/wiki/Piek%C5%82o_DLL*
Piekło DLL może objawiać się na różne sposoby; zwykle aplikacje się nie uruchamiają lub działają nieprawidłowo. Konkretna wersja biblioteki może być zgodna częścią wtyczek (i niezgodna z innymi), które jej wymagają.
Mogą wystąpić problemy w systemie Windows, jeśli DLL są o tej samej nazwie, ale różnych statusach wydania (datach i rozmiarach jednostek informacji) i mają być wykorzystywane przez wiele wtyczek.
Po drugie, DLL może zostać zaktualizowany przez nie wiadomo kogo, do innego programu, który stosuje wspólne biblioteki DLL, to powoduje wielki bałagan (stąd określenie 'DLL hell ').
Ludzie korzystający tylko z Pythona, są już w jego wersji 3.7, podczas gdy My GIMPersi, nadal używamy wersji 2.7.

Przykłady **dll** o tej samej nazwie, ale różnych statusach wydania:
GIMP wersja **Portable 64 bit**:
libgimpui-2.0-0.dll 2014-09-01 862 kB
GIMP 2.8.14 **64 bit** wersja stabilna (standalone)
libgimpui-2.0-0.dll 2014-08-26 142 KB

Istnieją również wtyczki wieloplikowe, takie jak np. **G'MIC** i **MathMap**, które używają np. pliku **dll** o tej samej nazwie, ale które różnią się rozmiarami jednostek informacji - w MathMap jest on większy.
Konkretnie jest to **libfftw3-3 .dll** (MathMap 2311KB, G`MIC 1606KB)
W takim konkretnym przypadku (jak i podobnych innych) można łatwo obejść problem
Należy umieścić wszystkie pliki G'MIC czy MathMap, w oddzielnych podkatalogach w strukturze katalogów GIMP-a.
U mnie jest to w strukturze katalogów: C:\Users\ Zbyma**.gimp-2.8**\plug-ins, a następnie w
Edycja/Preferencje/Katalogi/Wtyczki
wskazać tworzoną nową ścieżkę, czyli gdzie GIMP ma szukać dodatkowe wtyczki



Tak więc, wtedy jedna wtyczka nie koliduje z żadną inną.

Podkreślić należy również, że:
Twórcy Pythona raczej nie przejmują się tzw. zgodnością wsteczną - programy (polecenia, instrukcje) dopasowane do starszej wersji Pythona mogą nie działać w nowszej.
Z powodu częstych aktualizacji języka i braku zgodności wstecznej łatwo trafić na wtyczki niezgodne z wersją 2.7, a tym bardziej z najnowszą wersją Pythona 3.7. Trzeba się liczyć z koniecznością zmian w kodach źródłowych.
[O ile Python 2 jest bardzo podobny do Pythona 3, to Python 3 nie jest kompatybilny wstecz.
Znaczy to, że programy napisane pod Pythona 2, nie zawsze się uruchomią pod Pythonem 3.]
Instalatory pozwalają korzystać z większości wtyczek Pythona w GIMP-ie, (ale wtyczka może być swoistym wyjątkiem, ponieważ jest zbyt stara).
Podsumowując:
Jeśli więc np., kompilacja GIMP Portable Partha jest 64 bit i nie zawiera folderu 32bit, nie ma więc *dll* 32 bit.

Warto zwrócić uwagę na to:
GIMP 2.8.14 64 bit wersja stabilna (oficjalny instalator)

```
GIMP 2.8.14 Python Console
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.150
0 32 bit (Intel)]
>>>
```

także zainstalowana jest 32-bitowa wersja Python ale 2.7.5

GIMP 2.8.15 64 bit Portable Samj

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>>
```

Jest zainstalowana 32-bitowa wersja Python 2.7.6
Jeszcze jedno co bardzo komplikuje zadania,
niektóre z procedur dostępnych dla Python-Fu zmieniają się w sposób ciągły.
Przykład, pokazałem powyżej (`hello-world_Inter2.py`), wtyczka była pisana z przeznaczeniem do instalacji w GIMP 2.6, autor użył przykładowo procedury **pdb.gimp_image_add_layer**; podczas gdy obecnie do instalacji wtyczki w GIMP 2.8, powinna być zastąpiona procedurą **pdb.gimp_image_insert_layer** (z zmienioną liczbą argumentów!).

Pomoc mamy dostępną bezpośrednio na konsoli Python-Fu; przycisk Przeglądaj otwiera Przeglądarkę procedur, która dostarcza informacji na temat szukanych obecnie obowiązujących procedur dostępnych dla wtyczki. Dalej to tylko kwestia wykonania dopasowania poszczególnych procedur na poprawne dla danej wersji Python-Fu.

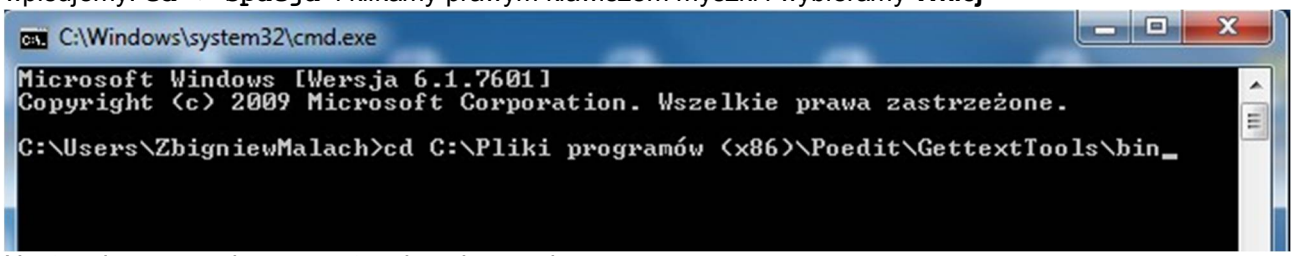
Z tych wywodów wynika uwaga ogólna:
Wtyczka opracowana i pracująca w GIMP i Python wersji 32-bit, nie musi pracować w GIMP 64-bit.

Dalej:
Udowodniono, że wtyczki GIMP-a, można zainstalować w dowolnym języku. Po instalacji możliwe jest również ich tłumaczenie na dowolny język. Umożliwiają to jak pokazano powyżej, pliki z tłumaczeniami o rozszerzeniach .mo i .po.

Ale czy jest metoda, aby można wykorzystać tylko posiadany plik **.mo**? Czyli:

Jak przerobić plik .mo na .po.

Metoda 1.
Za pomocą konsolowego programu można prze-konwertować pliki **.mo** na **.po**.
Aby to zrobić musimy uruchomić konsolę CMD, i przejść w niej do katalogu z instalacją PoEdit. Przeważnie ta komenda w Windows będzie wyglądała tak:
Najpierw ściągamy i instalujemy POEDIT
Jeśli mamy tylko plik **.mo** a nie **.po** ale chcemy dodać/edytować tłumaczenia, można 'zdekompilować' plik **.mo** do pliku **.po** z linii poleceń z zastosowaniem PoEdit, tzn. korzystając z Polecenia DOS
Otwieramy kolejno **C:\Pliki programów (x86)\Poedit\GettextTools\bin**
Teraz klikamy START => wpisujemy polecenie **`cmd`** i klikamy Enter, otworzy się ono DOS, w którym wpisujemy: **`cd + spacja`** i klikamy prawym klawiszem myszki i wybieramy **Wklej**

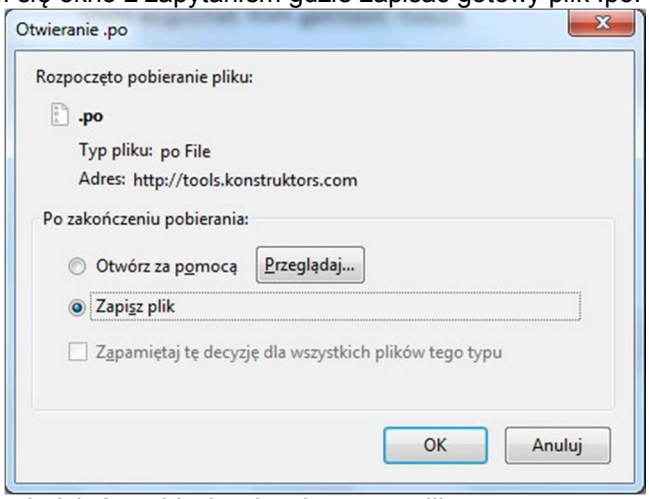


Następnie wprowadzamy następującą komendę:
`msgunfmt [path_to_file.mo] > [path_to_file.po]`
(**`msgunfmt`** [sciezka_do_pliku**.mo**] > [sciezka_do_zapisu_nowego_pliku**.po**])
Tak uzyskamy plik z rozszerzeniem **.po**, który możemy edytować programem PoEdit.
Ale to dla tych co lubią zabawy z DOS.

Metoda 2.
(Szybka metoda dla wygodnych)
Czy jest inny sposób, aby przekonwertować plik **.mo** do źródłowego pliku **.po**, gdy plik **.po** nie jest dostępny?
Najprościej mamy możliwość wykonać to *narzędziem online*.
Tak jak powyżej najpierw tworzymy (*najprościej i najbezpieczniej*) *na Pulpicie* folder roboczy w którym umieszczamy skopiowany interesujący Nas plik tłumaczenia wtyczki **.mo**,
Teraz otwieramy stronę :

http://tools.konstruktors.com/

i wskazujemy ścieżkę do Naszego folderu roboczego, **Przeglądaj…** zaznaczamy Nasz plik i klikamy **Otwórz** i następnie **Convert**, pojawi się okno z zapytaniem gdzie zapisać gotowy plik .po:



I to już wszystko, można powiedzieć, że błyskawicznie mamy plik .po..

Powyżej stwierdziłem ,
" każda wtyczka do tłumaczenia **musi** korzystać z unikalnej nazwy - ***adresu domeny***, pod którą można umieścić folder. To jest identyfikator.

Który zawiera: \pl\LC_MEESAGE

foldery językowe np. pl, zawierają podfoldery LC_MESSAGES (zmienna środowiskowa), które z kolei zawierają pliki **xyz.mo**.
Każdy LC_MESSAGES zawiera zestaw tłumaczeń różnych wtyczek na dany język."

Tyle teoria, a jak to się ma do rozwiązań praktycznych?.

Zaintrygowało mnie, że:
znalazłem szereg wtyczek, w języku angielskim, które zostały przez autora umiędzynarodowione (otwierając w Notepad++ wtyczkę, widzimy funkcję **_ ( )** przy niektórych ciągach), ale **wszystkie** one mają podaną

***Nazwę** Domeny tłumaczenia,* jako **"gimp20-python".**

Zacząłem więc szukać wyjaśnienia i na stronach GIMP-a nic nie znalazłem, wobec powyższego szukałem własnego rozwiązania zagadnienia.

Mamy przykładowo wtyczkę: **python-fu-foggify**

```
#!/usr/bin/env python

#   Gimp-Python - allows the writing of Gimp plugins in Python.
#   Copyright (C) 1997  James Henstridge <james@daa.com.au>
#
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 2 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
#   You should have received a copy of the GNU General Public License
#   along with this program; if not, write to the Free Software
#   Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

from gimpfu import *
import time

gettext.install("gimp20-python", gimp.locale_directory, unicode=True)

def foggify(img, layer, name, colour, turbulence, opacity):

    gimp.context_push()
    img.undo_group_start()

    if img.base_type is RGB:
        type = RGBA_IMAGE
    else:
        type = GRAYA_IMAGE
    fog = gimp.Layer(img, name,
                     layer.width, layer.height, type, opacity, NORMAL_MODE)
    fog.fill(TRANSPARENT_FILL)
    img.add_layer(fog, 0)

    gimp.set_background(colour)
    pdb.gimp_edit_fill(fog, BACKGROUND_FILL)

    # create a layer mask for the new layer
    mask = fog.create_mask(0)
    fog.add_mask(mask)
```

```
    # add some clouds to the layer
    pdb.plug_in_plasma(img, mask, int(time.time()), turbulence)

    # apply the clouds to the layer
    fog.remove_mask(MASK_APPLY)

    img.undo_group_end()
    gimp.context_pop()

register(
    "python-fu-foggify",
    N_("Add a layer of fog"),
    "Adds a layer of fog to the image.",
    "James Henstridge",
    "James Henstridge",
    "1999,2007",
    N_("_Fog..."),
    "RGB*, GRAY*",
    [
        (PF_IMAGE, "image",       "Input image", None),
        (PF_DRAWABLE, "drawable", "Input drawable", None),
        (PF_STRING, "name",       _("_Layer name"), _("Clouds")),
        (PF_COLOUR, "colour",     _("_Fog color"),  (240, 180, 70)),
        (PF_SLIDER, "turbulence", _("_Turbulence"), 1.0, (0, 10, 0.1)),
        (PF_SLIDER, "opacity",    _("Op_acity"),    100, (0, 100, 1)),
    ],
    [],
    foggify,
    menu="<Image>/Filters/Render/Clouds",
    domain=("gimp20-python", gimp.locale_directory)
    )

main()
```

W **GIMP 2.8.14 64 bit** z oficjalnego instalatora, oraz przykładowo w **GIMP 2.8.15 64 bit Portable kompilacji Samj** znajdziemy ją w: **`Filtry/Renderowanie/Chmury/Fog...`** – a plik **foggify.py** jest w **`lib/gimp/2.0/plug-ins`**

W **GIMP 2.8.14 64 bit** z oficjalnego instalatora, interaktywne okno wtyczki, pojawia się nieprzetłumaczone,

Chociaż zbiorcze tłumaczenie dla kilku wtyczek umieszczone jest w pliku "gimp20-python", który za pomocą http://tools.konstruktors.com/ zdekompilowałem do **.po** i odczytałem, że ciągi tłumaczeń na j. polski dla wtyczki są zawarte w pliku

```
msgid "_Fog color"
msgstr "Kolor _mgły"

msgid "_Fog..."
msgstr "_Mgła..."

msgid "_Layer name"
msgstr "Nazwa _warstwy"
```

Podgląd wycinek zrzutu z Notepad++

Wobec tego sprawdziłem jeszcze umieszczając plik wtyczki w **GIMP 2.8.6 Portable 32 bit Partha** w **/.gimp-2.8/plug-ins,** (plik "gimp20-python", jest standardowo zawarty w kompilacji), otrzymano wynik:

Jak widać nawet "Etykieta" wtyczki jest przetłumaczona, oraz:

opisy w oknie przetłumaczone.

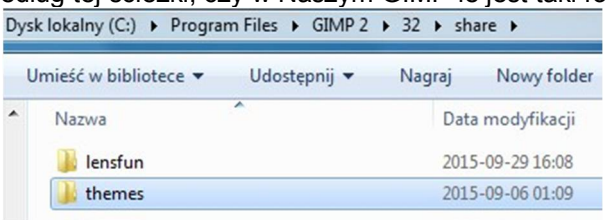
Wniosek z tego jest taki, że niektórzy autorzy, nie stosują się do zaleceń **gettext**, co w praktyce prowadzi do tego że trudno jest stwierdzić czy w kompilacji jest tłumaczenie danej wtyczki na Nasz język (lub inny) i gdzie je zlokalizować !!!. Ale teraz już wiemy gdzie szukać tych plików, tylko trudno poszukiwać ciągów należących do konkretnej wtyczki. Zauważyć można że są one w układzie alfabetycznym (wg. j. angielskiego).

Ale nie możemy się poddawać, trzeba spróbować ustalić, co jest przyczyną takiej sytuacji:

Przede wszystkim, gdy mamy wątpliwości z instalacją wtyczki, lub wprowadzaliśmy jakiejś zmiany po jej zainstalowaniu w jej **register**, należy w \.**gimp-2.8**\ usunąć plik **pluginrc** – ten plik może być bezpiecznie usunięty i będzie automatycznie wygenerowany przez GIMP-a, badającego zainstalowane wtyczki.
Jeśli taki zabieg nie pomaga po uruchomieniu GIMP-a, otwieramy plik pluginrc np. w WordPad, plik jest plikiem tekstowym. Szukamy interesującego nas wpisu dotyczącego określonej wtyczki.
Dane są uszeregowane alfabetycznie, przy czym litera **a** jest na samym dole wyświetlonego pliku.
Jakie tam znajdziemy dziwne ścieżki do **gimp.locale_directory.**
W tym konkretnym przypadku interesuje Nas przykładowo, dlaczego w **GIMP 2.8.14 64 bit** wtyczka **foggify.py** pojawia się nieprzetłumaczona, pomimo, że jak stwierdziliśmy w zbiorczym pliku tłumaczeń "gimp20-python", to tłumaczenie istnieje.
Odszukano zapis w pluginrc dla tej wtyczki:

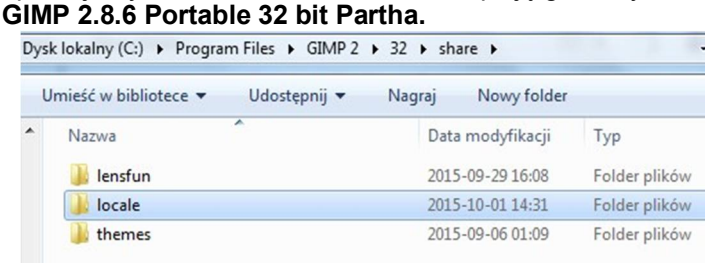
```
(plug-in-def "C:\\Program Files\\GIMP 2\\lib\\gimp\\2.0\\plug-ins\\foggify.py" 1409083666
    (proc-def "python-fu-foggify" 1
         "Add a layer of fog"
         "Adds a layer of fog to the image."
         "James Henstridge"
         "James Henstridge"
         "1999,2007"
         "_Fog..."
         1
        (menu-path "<Image>/Filters/Render/Clouds")
        (icon stock-id -1 "")
         "RGB*, GRAY*"
         7 0
        (proc-arg 0 "run-mode" "The run mode { RUN-INTERACTIVE (0), RUN-NONINTERACTIVE (1) }")
        (proc-arg 13 "image" "Input image")
        (proc-arg 16 "drawable" "Input drawable")
        (proc-arg 4 "name" "Layer name")
        (proc-arg 10 "colour" "Fog color")
        (proc-arg 3 "turbulence" "Turbulence")
        (proc-arg 3 "opacity" "Opacity"))
    (locale-def "gimp20-python" "C:\\Program Files\\GIMP 2\\32\\share\\locale"))
```

Kolorem czerwonym wyróżniono ostatni wers, który jest ścieżką bezwzględną w Naszym GIMP-ie, do pliku tłumaczenia umieszczonego przez autora w "gimp20-python".
Wobec czego sprawdzam, według tej ścieżki, czy w Naszym GIMP-ie jest taki folder:



A więc sprawa się wyjaśniła, nie ma takiego folderu w Naszym GIMP-ie z Oficjalnego instalatora.
Powyżej pisałem o ***Piekle DLL,*** które może objawiać się na różne sposoby; zwykle aplikacje się nie uruchamiają lub działają nieprawidłowo. Oraz wspominałem, że może zostać zaktualizowany przez nie wiadomo kogo, do innego programu, który stosuje wspólne biblioteki DLL, to powoduje wielki bałagan (stąd określenie 'DLL hell '). Okazuje się, że bałagan nie tylko z dll..

Aby rozwiązać problem spróbujemy zainstalować taki folder, skopiuję go z używanego również (jak pokazywałem powyżej) **GIMP 2.8.6 Portable 32 bit Partha.**



Tak jak wspomniałem, teraz usuwam plik pluginrc i uruchamiam GIMP-a.
Pojawia się przetłumaczone, interaktywne okno wtyczki:

Przy okazji sprawdziłem teraz również okno wtyczki hello_world

I to by było na tyle wprowadzenia, resztę trzeba opanować samemu!

Jak pokazano powyżej, proces lokalizacji nie jest niczym skomplikowanym, ale oczywiście wszelkie zmiany jakie zostaną wykonane według tego poradnika są na Waszą odpowiedzialność. Pamiętajcie więc o kopiach zapasowych plików, które będziecie zmieniali.
W razie jakichkolwiek problemów bardzo to Wam ułatwi naprawę tego, co popsujecie!.

Dobrzy programiści, ucząc się poprzez modyfikację istniejących wtyczek, często są w stanie osiągnąć ciekawe rzeczy po zaledwie kilku dniach pracy.

PS.

Zainteresowani mogą zapoznać się również z:
http://www.gimpuj.info/index.php?topic=56802.0 **02.07.2012**
i dalej:
https://l10n.gnome.org/POT/gimp.master/gimp-plug-ins.master.pot
"pot" – już mamy teraz tylko dalszy wysiłek.

Autor opracowania:
inż. Zbigniew Małach
Zbyma72age