Poradnik

Wtyczki (Rozszerzenia) Python-Fu do GIMP-a.

(Wprowadzenie do podstaw programowania, czytania i spolszczania)

Dodatek (Plugin) Python-Fu

18.10.2015r

Motto: Czas prędzej płynie, kiedy głowa jest czymś zajęta.

Podstawowym zadaniem poradnika jest nie tyle nauczenie pisania wtyczek, ale przede wszystkim ułatwienie czytania już istniejących wtyczek i ewentualne ich przetłumaczenie.

Wstęp

W GIMPie nie tylko można edytować obrazy, ale również pisać dodatki, które te obrazy będą tworzyć. Dodatki można *pisać* za pomocą języka skryptowego Scheme (który już opisałem: *Zbyma_Wprowadzenie_do_pisania_Script-Fu_w_GIMPver1.pdf* http://1drv.ms/1KMgsdl) oraz za pomocą kompilowanego języka programowania Python, który jest zaliczany do języków **zorientowanych obiektowo**.

(W polskiej wersji GIMP-a nazwa Plug-in => Dodatek jest przetłumaczona jako **Wtyczka**.)

Dodatkowo: http://pl.wikibooks.org/wiki/GIMP/Programowanie

Dodatkowe funkcje w GIMP-ie można realizować poprzez :

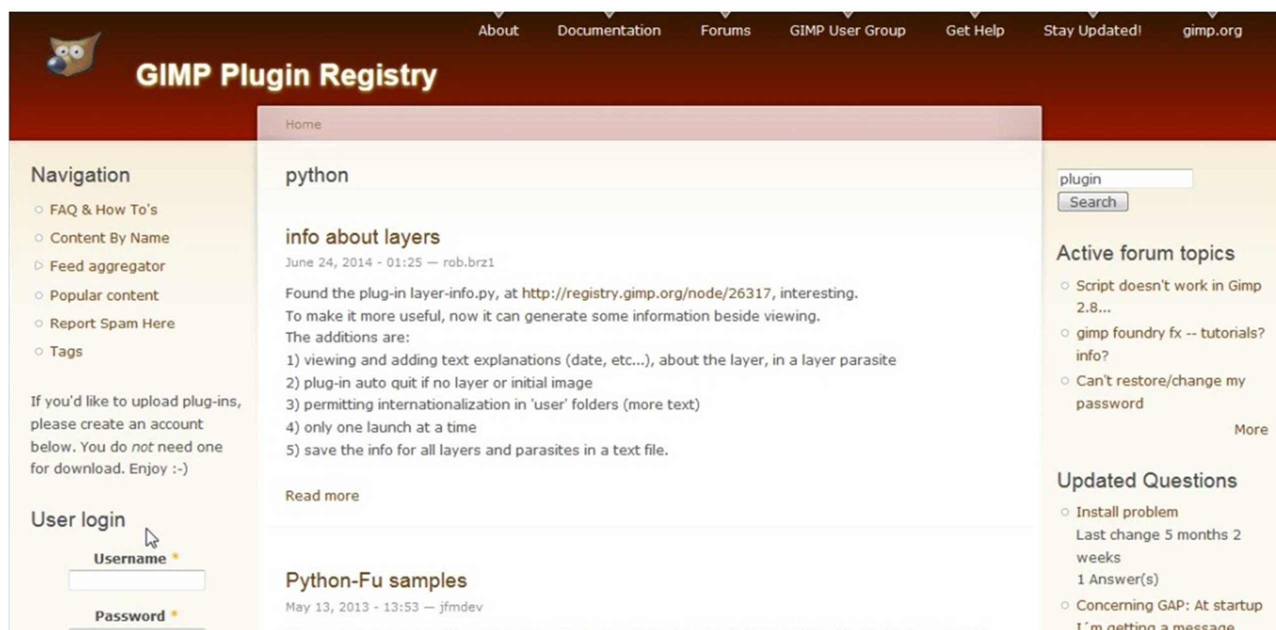
- wtyczki (**ang**. *plugin*): osobny proces wywoływany przez rdzeń GIMP-a , kod źródłowy wymaga kompilacji, zmieniają obrazy (**Python**, rozszerzenie *.**py**, mimo że są to skrypty to wymagają kompilacji).
- rozszerzenia (**ang**. *extension*): osobny proces, kod źródłowy wymaga kompilacji, nie zmieniają obrazów
- skrypty (**ang**. *script*): nie wymaga kompilacji, zmienia obraz

Poradnik zostanie poświęcony jedynie językowi kompilowanemu Python oraz modułowi Python-Fu. Poradnik odnosi się do środowiska Windows 7 oraz GIMP Portable w których pracuję. Instalacja jest zależna od tego, którą wersją GIMP-a będziemy dysponować. Najlepszym sposobem jest instalacja **GIMP *Portable*** wersji 2.8.x, który już ma Pythona 2.7.6. Podczas procesu instalacji standardowej wersji GIMP-a, trzeba wybrać pełną instalacją i zaznaczyć, by zainstalowano również moduł Python`a do GIMPa, Python-Fu.
Inforacje na ten temat:

http://www.gimpusers.com/tutorials/install-python-for-gimp-2-6-windows.html
http://www.mahvin.com/2009/09/gimp-how-to-install-scripts-plug-ins-to.html
http://www.gimpuj.info/index.php/topic,26623.0.html

Dodatek (plug-in) przeciw skryptowi

Częso nie jest jasne, jaka jest różnica między nimi. Co stanowi skrypt i co stanowi wtyczkę (dodatek)? Większość istniejących dodatków wraz z kodem źródłowy jest udostępniana za pomocą witryny *GIMP Plugin Registry*. Po wyświetleniu witryny GIMP Plugin Registry pojawi się strona podobna do pokazanej poniżej:



Stronę można przeglądać na różne sposoby zgodnie z **Navigation**.

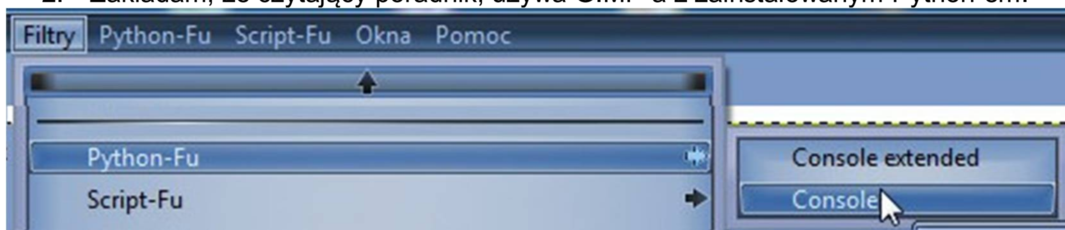
W przypadku dodatków spotykamy się z dwoma ich podstawowymi typami: ***skryptami i dodatkami***. Ogólnie rzecz biorąc, dodatek jest niewielkim programem, który rozszerza funkcjonalność zasadniczego programu jakim jest GIMP.
Programiści pisząc programy używają w tym celu różnych rodzajów języków programowania. Języki te można zaliczyć do dwóch podstawowych kategorii: ***języki skryptowe i języki kompilowane***.

Domyślnie GIMP obsługuje języki **Python**, **Perl** i **Scheme**.
Język skryptowy jest łatwo czytelny i każdy wiersz kodu jest dynamicznie **iterpretowany** przez komputer (wiersz po wierszu). Ponieważ skrypty działają w sposób dynamiczny nie zawsze są zoptymalizowane i wymagają więcej czasu na proces przetwarzania. Wtyczki Python GIMP-a są wtyczkami interpretowanymi.
Natomiast kod źródłowy utworzony za pomocą kompilowanego języka programowania, przed uruchomieniem jest konwertowany do postaci języka komputera, czyli zer i jedynek.
Proces konwersji, nazywany ***kompilowaniem***, powoduje że kod *nie jest czytelny dla człowieka*, ale jest łatwiejszy do zrozumienia dla komputera, ma on miejsce dużo wcześniej przed uruchomieniem programu.

Bardziej poglądowo:
Proces interpretowania, można porównać do bierzącego tłumaczenia słów wypowiadanych do innej osoby z którą prowadzimy rozmowę, zabiera to więcej czasu, bo tłumacz musi przed przetłumaczeniem, zaczekać aż zakończymy wypowiedź.
Natomiast w przypadku procesu kompilowania, możemy wyobrazić sobie, że chcemy przekazać informację jakiejś osobie, która posługuje się innym językiem. Jeśli wcześniej wiemy co chcemy powiedzieć, przekazujemy tą informację ekspertowi, który przetłumaczy poszczególne słowa, po czym możemy nagranie odtworzyć innej osobie.
Na potrzeby konfiguracji i instalacji w GIMP-ie, dodatek napisany w *języku skryptowym* jest nazywany skryptem, natomiast dodatek utworzony za pomocą *języka kompilowanego* jest faktycznym dodatkiem.
Wtyczki GIMP napisane w języku **C** są skompilowane i połączone z bibliotekami GIMP-a. Mogą one uzyskać dostęp do funkcji w tych bibliotekach lub funkcji w PDB. Są one rozprowadzane jako pliki wykonywalne z oznaczeniem "**.exe**".
Trochę w: Compiling Gimp for Windows with VirtualBox - http://gimpchat.com/viewtopic.php?f=23&t=8111

Uwaga:

1. W poradniku będę czsami zamiennie używał określenia dodatek i wtyczka.
2. Zakładam, że czytający poradnik, używa GIMP-a z zainstalowanym Python-em.



Jeśli tak, to wtedy widzimy.

Python-Fu to rozszerzenie skryptów dla GIMP-a, podobne jak Script-Fu.
Główną różnicą jest to, co w pierwszej kolejności się wzywa.
W Script-Fu, skrypt-fu wykonuje skrypt, podczas gdy w GIMP-Python wtyczka jest sterowaniem.
Rzeczywiście, możemy zauważyć, że w GIMP-Python początkiem skryptu jest linia:

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

W ten sposób system będzie wiedział, że musi znaleźć lokalizację Pythona za pomocą programu **`env`** i dopiero wtedy użyć go jako interpretera.
A potem wtyczka do GIMP jest ładowana poleceniem **`import`**.
Kolejnym punktem różnicy pomiędzy GIMP-Python i Script-Fu jest to, że GIMP-Python przechowuje zdjęcia, warstwy, kanały i inne typy, jako obiekty, a nie tylko przechowuje ich **ID** (jak to jest w Script-Fu).
To pozwala na lepszą kontrolę typu, czego brakuje w Script-Fu, i pozwala tym typom działać jako obiekty, kompletnie wraz z atrybutami i metodami.
Ponadto, GIMP-Python nie jest ograniczony do zaledwie wywoływania procedur z PDB. Realizuje również resztę **`libgimp`**, w tym kafelków i regionów pikseli, a także dostęp do innych funkcji niższego poziomu.

Informacja historyczna
Python stworzony został w latach 90. przez Guido van Rossuma, nazwa zaś pochodzi od tytułu serialu komediowego emitowanego w BBC pt. “Latający cyrk Monty Pythona”.

Pobieżne wprowadzenie do Pythona

Wersję Pythona oznaczoną numerem 2.7.10 wydano 23 maja 2015 r.

Procedury i funkcje

to podprogramy, stanowiące pewną całość, posiadające jednoznaczną nazwę i ustalony sposób wymiany informacji z pozostałymi częściami programu.
Są stosowane do wykonania czynności wielokrotnie powtarzanych przez dany program.

Różnice między funkcją a procedurą:

- sposób przekazywania wartości
- odmienne sposoby wywołania

- zadaniem procedury (podprogramu) jest wykonanie pewnej sekwencji czynności (poleceń), polegających zwykle na obliczaniu jednej lub wielu wartości
  Liczba, kolejność i typy parametrów muszą być zgodne z definicją procedury!
- zadaniem funkcji jest przekazywać pewne parametry jej wykonania, jak również może zwracać wynik jej działania.
  Liczba, kolejność i typy parametrów oraz typ wartości zwracanej muszą być zgodne z definicją funkcji!

Parametry procedury wymienione w nawiasie na początku jej ciała nazywamy parametrami (argumentami) formalnymi,
wartości dla parametrów formalnych podane w wywołaniu procedury nazywamy parametrami aktualnymi (argumentami wywołania) procedura może być także bezparametrowa.
Nazwa procedury musi być identyfikatorem *(w przypadku funkcji wyjątkiem są nazwy operatorów)* procedura nie zwraca wartości *funkcja zwraca dokładnie jedną wartość.*

Procedura może mieć swoje własne zmienne (zmienne lokalne), deklarowane w jej części deklaracyjnej zmienna lokalna jest widziana tylko wewnątrz procedury z tego powodu zmienne lokalne różnych procedur mogą mieć takie same nazwy **zmienne lokalne procedury to nie to samo co jej parametry.**

Kalkulator:

Uwaga: Publikowane poniżej ekranowe zrzyty pochodzą z **Konsola Python-Fu**

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> type(5)
<type 'int'>
>>> type(5.0)
<type 'float'>
>>> 5/3
1
>>> 5.0/3
1.6666666666666667
>>> 5/3.0
1.6666666666666667
>>> 5./3
1.6666666666666667
>>> |
```

`type` liczb, `"int"` – całkowite i `"float"` – zmiennoprzecinkowe

`type(1) => <class 'int'>`

`type(1.0) => <class 'flooat'>`

Operatory matematyczne:

- Jeżeli liczby **x** i **y** są typu całkowitego, to wynik dzielenia też jest typu całkowitego. Zaokrąglanie jest zawsze (również dla liczb ujemnych) w dół. (tutaj / - operator dzielenia)

```
3/2 => 1
(-3)/2 => -2
```

Jeżeli potrzebny jest zmiennoprzecinkowy wynik dzielenia liczb typu całkowitego, to trzeba:

```
3./2 => 1.5
float(3)/2 => 1.5
```

- Wynik dzielenia x // y (dzielenie całkowite) ma zawsze wartość całkowitą, jeżeli jedna z liczb **x** lub **y** jest typu **float**, to wynik też jest typu **float**. Zaokrąglanie jest zawsze (również dla liczb ujemnych) w dół.

```
3.23 // 2.11 => 1.0
-3.23 // 2.11 => -2.0
3 // 2 => 1
```

- Operacja **x % y** (reszta z dzielenia) jest określona dla wszystkich rodzajów liczb (dodatnie, ujemne, zespolone, całkowite, zmiennoprzecinkowe). W mających praktyczne znaczenie przypadkach: **x** i **y** nie są zespolone, **y > 0** zachodzi równość **`x = y*(x//y) + x % y`**.

```
4 % 3 => 1
-4 % 3 => 2
4.567 % 1.103 => 0.15500000000000025
```

- Argument **n** funkcji **round** może być ujemny.
  [**round** - zaokrąglenie do **n** miejsc po przecinku **`round(x,[n=0])`**]

```
round(1457, -1) => 1460.0
round(1457, -2) => 1500.0
```

http://bogdan.students.wmi.amu.edu.pl/education/python%20i%20django/Wyklad_4.html#1 http://karnet.up.wroc.pl/~jasj/cwiczenia/kwpp5.html

Kolejność wykonywania

Przykładowo, operator mnożenia ma wyższy priorytet (**ang.** *precedence*) od operatora dodawania.
O kolejności wykonywania operacji decyduje tabela priorytetów operatorów.
Tabela daje nam spojrzenie na priorytety operatorów w Pythonie, od najwyższego (wykonywanego najpierw) do najniższego, wykonywany w ostatniej kolejności, pochodzi z dokumentacji Pythona: https://docs.python.org/2/reference/expressions.html#evaluation-order . Priorytety operatorów są dobrane tak, by były zgodne z przyzwyczajeniami wziętymi z matematyki. Operatory są zazwyczaj łączone od lewej do prawej, co oznacza, że operacje z tym samym pierwszeństwem są wykonywane od lewej do prawej. **Przykładowo** 2 - 3 - 4 jest wykonywane jak (2 - 3) - 4, czyli daje wynik −5.
Wyjątkiem w stosunku do łączności lewostronnej jest operator potęgowania, który jest łączony od prawej do lewej,

Łączenie łańcuchów znaków oraz formatowanie łańcucha znaków.

Formatowanie jest jedną z wielu możliwości połączenia łańcuchów znaków, jednak formatowanie łańcucha znaków nie jest tym samym co łączenie łańcuchów.
Łączenie łańcucha znaków możemy wykonywać jedynie na innych łańcuchach.

Łańcuch znaków:

http://python.edu.pl/byteofpython/2.x/05.html#lancuchy

Przed przystąpieniem do tworzenia wtyczek za pomocą konsoli należy przypomnieć:
>>>
String to nazwa typu danych służącego do przechowywania **napisów** (zmiennych łańcuchowych).
Angielski **String** znaczy dosłownie ***ciąg***, *sznur*, więc jednym ze sposobów w jakim można mówić o tym typie to **ciąg znaków**. Innymi określeniami są **łańcuch znaków** lub po prostu **łańcuch** czy też **napis**.

`String => str`

Python może manipulować ciągami znaków (napisami), które można wyrazić na parę sposobów. Mogą one być objęte pojedynczym lub podwójnym cudzysłowem.
Za pomocą potrójnych cudzysłowów `"""` lub apostrofów `'''`możemy oznaczać łańcuchy wielolinijkowe. Co więcej, wewnątrz nich możemy swobodnie używać zarówno pojedynczych, jak i podwójnych cudzysłowów.
https://pl.wikipedia.org/wiki/Cudzys%C5%82%C3%B3w

Ciągi znaków, są zamknięte w doublequotes (") **prosty** cudzysłów (podwójny apostrof).
Aby utworzyć ciąg tekstowy, używamy prostego, a **nie** cudzysłowu drukarskiego (”). Wszystkie znaki pomiędzy nimi są interpretowane jako dane tekstowe czyli napisy.
Stosowanie w konsoli cudzysłowu drukarskiego kończy się wyświetleniem błędu.
Ważne jest, aby ciąg kończył się tym samym znakiem, jakim się rozpoczął.
Długość łańcucha jest liczbą znaków, które zawiera. Pierwszy znak łańcucha ma indeks 0, drugi 1 itd..
Konsola (Interpreter) drukuje (wyświetla) wynik działań na napisach w ten sam sposób, w jaki są one wprowadzane: objęte cudzysłowami i wraz z cudzysłowami i innymi *zabawnymi* znakami poprzedzonymi znakiem ukośnika (\), tak aby w sposób dokładny pokazać zawartość ciągu.
Łańcuch objęty jest cudzysłowem, jeśli zawiera tylko poroste cudzysłowy.

Jak poprawnie wpisać prosty apostrof (') i podwójny apostrof (**"**) do skryptu lub dodatku w Wordzie?

Przy rozmiarze czcionki 10 lub 12 różnica pomiędzy cudzysłowem prostym a drukarskim, jest niewidoczna, ale po zastosowaniu może okazać się nieprzyjemną pomyłką (czego sam doświadczyłem).
Jak zobaczyć różnicę? Wpisujemy np. podwójny cudzysłów. Teraz zaznaczamy ten znak i powiększamy kilkakrotnie używając skrótu (**Ctrl** i zamykający nawias kwadratowy **]**). Jeśli zobaczymy
”
mamy ustawiony w Word cudzysłów drukarski (a więc także taki apostrof).
Podczas wpisywania tekstu w niektórych programach pakietu Microsoft Office cudzysłowy proste (') lub podwójne (") są automatycznie zamieniane na *cudzysłowy drukarskie*. Tę funkcję można w **Word** włączyć lub wyłączyć.

1. W **Word** Klikamy **przycisk**, a następnie na dole, klikamy przycisk **Więceń poleceń ….**
2. Klikamy przycisk **Sprawdzanie**, a następnie klikamy polecenie **Opcje Autokorekty**.
3. W oknie dialogowym **Autokorekta** wykonamy następujące czynności:
   - Klikamy kartę **Autoformatowanie podczas pisania** i w obszarze **Zamień podczas pisania** zaznaczamy lub czyścimy pole wyboru **Cudzysłowy "proste" na „drukarskie”**.
   - Klikamy kartę **Autoformatowanie** i w obszarze **Zamień** wstawiamy zaznaczenie lub czyścimy pole wyboru **Cudzysłowy "proste" na „drukarskie”**.

Musi być "**proste**", inaczej Konsola wyświetli "**Error:…**".

Każdy znak w łańcuchu ma swój indeks.
Przez indeksowanie stringów rozumiemy dostęp do wskazanych numerycznie elementów łańcucha znaków, czyli jego podzbiorów jedno jak i wieloelementowych. Python ma dość rozbudowaną możliwość dostępu do części stringa po indeksach znaków.
Pierwszy znak w łańcuchu ma ideks **0**. Warto jednak dodać, że indeks może przyjąć również wartości ujemne, co oznacza znak liczony od końca łańcucha.
Możemy również wskazać zakres indeksów, które w rezultacie dadzą nam podzbiór całości. Ujemne wartości indeksów podobnie jak powyżej, oznaczają położenie liczone od końca stringa.
>>>

Linie logiczne i fizyczne poleceń
Linia fizyczna to to, co *widzimy*, kiedy piszemy program.
Linia logiczna to to, co Python *widzi* jako pojedyncze polecenie.
Python domyślnie zakłada, że każda linia *fizyczna* odpowiada *logicznej*.
Przykładem linii logicznej jest szeroko znane `print 'Hello, World!'`.
Jeżeli znajdowało się ono również w jednej linii w programie (tak, jak to widzimy w edytorze), to była to również jedna linia fizyczna.
Wskazanym jest używać zawsze jednego polecenia na linię, dzięki czemu kod staje się bardziej czytelny.
Jeśli chcemy zawrzeć więcej, niż jedną linię logiczną w jednej linii fizycznej, możemy użyć **średnika (;)**, który oznacza po prostu koniec linii logicznej/polecenia.
Przykładowo równoznaczne zapisy:

```
>>> i = 5;
>>> print i;
5
>>> i = 5
>>> print i
5
>>> i = 5; print i
5
>>> i = 5; print i
5
>>> |
```

Jednakże mimo tak dużej liczby możliwości, **zalecanym jest** przyzwyczajenie się do **pisania wyłącznie pojedynczych linii logicznych w pojedynczych liniach fizycznych**.
Tylko w przypadku naprawdę długich linii logicznych możemy je zapisać w kilku liniach fizycznych.
Chodzi o to, **aby unikać użycia średnika** *tak bardzo*, jak to tylko możliwe, ponieważ to skutkuje dużo czytelniejszym kodem.
Żeby być szczerym, *bardzo rzadko widziałem* średnika we wtyczce Python-fu.
Poniżej jest przykład zapisu jednej linii logicznej **napisu** w wielu liniach fizycznych.

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> #!/usr/bin/env python
>>> # -*- coding: utf-8 -*-
>>> # Nazwa pliku: zmienne.py
>>>
>>> i = 5
>>> print i
5
>>> i = i + 1
>>> print i
6
>>> s = '''To jest łańcuch wielolinijkowy.
... To jest drugi wiersz.'''
>>> print s
To jest łańcuch wielolinijkowy.
To jest drugi wiersz.
>>> |
```

(Instrukcja `print`, może zostać użyta do wypisywania ciągów znaków bez okalających je cudzysłowów lub znaków cytowania).

`Type('Hallo') => <class 'str'>`

Jak to działa:
Najpierw przypisaliśmy stałą dosłowną **5** do zmiennej **`i`** za pomocą operatora przypisania (**=**).
Linia ta nazywa się poleceniem, ponieważ zleca ona Pythonowi wykonanie czegoś: w tym przypadku łączymy nazwę zmiennej **`i`** z wartością **5**.
Następnie, również za pomocą polecenia, wypisujemy wartość zmiennej **`i`** na ekran, używając **`print`**.
Później dodajemy **1** do wartości przechowywanej w zmiennej **`i`** i zapisujemy nowo obliczoną wartość do tej zmiennej. Potem wypisujemy wartość zmiennej **`i`**, dostajemy **6**.
Analogicznie postępujemy ze zmienną **`s`**, której przydzielamy wartość dosłowną łańcucha i wypisujemy ją..

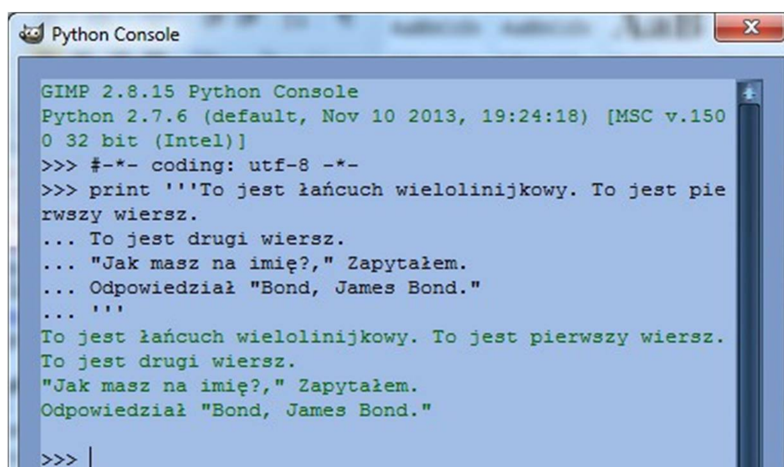
`"To jest przykładowa wtyczka wykonana przez Zbyszka dla chętnych poznania podstaw Pythona i pisania Python-Fu"`
`"dzień dobry"` i `'dzień dobry'` są sobie równoważne; ale `'dzień dobry"` jest błędem – trzeba być konsekwentnym, na koncu napisu musi wystepowac ten sam znak cytowania, co na poczatku.

Łączenie łańcuchów znaków

Łańcuchy znaków => Ciągi napisów => str mogą być ujęte w potrójnych cudzysłowach: `"""` lub apostrofach `'''(znaki cytowania),a` wewnątrz nich możemy swobodnie używać zarówno apostrofów, jak i cudzysłowów:

`""" Witaj w świecie Pythona"""`

Ale również:



(Instrukcja `print`, może zostać użyta do wypisywania ciągów znaków bez okalających je cudzysłowów lub znaków cytowania).

```
print "jeden\ndwa\ntrzy"
```

Jeśli **napis** rozciąga się na wiele wierszy, można również *znak nowej linii* **zacytować** za pomocą znaku ukośnika (***znak ucieczki***) **\n**:

```
#-*- coding: utf-8 -*-
s = "Jest to raczej długi ciąg znaków zawierający\n\
kilka linii tekstu.\n\
    Należy zauważyć, że znaki spacji i znaki białe na początku linii\n\
 są znaczące."
print s
```

Kliknięcie **Enter** spowoduje pojawienie się zaraz za tym:

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
>>> #-*- coding: utf-8 -*-
>>> s = "Jest to raczej długi ciąg znaków zawierający\n\
... kilka linii tekstu.\n\
...     Należy zauważyć, że znaki spacji i znaki białe na początku linii\n\
...  są znaczące."
>>> print s
Jest to raczej długi ciąg znaków zawierający
kilka linii tekstu.
    Należy zauważyć, że znaki spacji i znaki białe na początku linii
 są znaczące.
>>> |
```

[**Białymi znakami** są nazywane te symbole, które są używane w tekście i nie posiadają swojej reprezentacji graficznej. Przykładem takiego znaku jest spacja, tabulacja czy też znak przejścia do nowej linii (Enter).]

Ale również:
Jeżeli chcemy wewnątrz łańcucha, zawrzeć *pojedynczy cudzysłów* np.

'Nie lubię Harry'ego'

musimy zastosować zapis tego *pojedynczego cudzysłowu* za pomocą znaku ucieczki \'.

'Nie lubię Harry\'ego'

Inny przypadek:

```
"To jest pierwsze zdanie. \
A to drugie."
```

Kliknięcie **Enter** spowoduje pojawienie się zaraz za tym:

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> "To jest pierwsze zdanie. \
... A to drugie."
'To jest pierwsze zdanie. A to drugie.'
>>> |
```

Jak widać powyższy przykład jest równoważny z: "To jest pierwsze zdanie. A to drugie."
Musimy umieścić spację przed ukośnikiem wstecznym, bo w przeciwnym wypadku **A** występowałoby w łańcuchu bezpośrednio po kropce.
Oprócz **\n** istnieje jeszcze wiele podobnych znaków specjalnych, szczególnie użyteczny może być znak tabulacji: **\t**.

Czasami sytuacja jest tak jednoznaczna, że nie trzeba używać ukośników wstecznych. Dzieje się tak w przypadku, gdy w linii logicznej są nawiasy okrągłe, kwadratowe lub klamrowe. Takie sytuacje zobaczymy w **listach**.

Dalej:

```
# "To jest łańcuch znaków komentarza"
```

ale:

```
"# To nie jest komentarz. To jest łańcuch znaków."
```

Można tak:

```
a = 'Witaj w swiecie Pythona'
b = ' w 2015r'
a+b
'Witaj w swiecie Pythona w 2015r'
```

Łańcuchy mogą być sklejane za pomocą operatora **+** a powielane za pomocą *****:

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> "Ala" + "ma" + "kota"
'Alamakota'
>>> "Ala" + " ma" + " kota"
'Ala ma kota'
>>> "Witaj w swiecie Pytkhona" + " w 2015r"
'Witaj w swiecie Pytkhona w 2015r'
>>> |
```

Jeżeli umieścimy łańcuchy obok siebie, Python automatycznie je połączy. Przykład:

```
>>> 'Ja mam' ' tego' ' po' ' uszy'
'Ja mam tego po uszy'
>>> |
```

Przykład skutku umieszczenia znaku spacji w 2, 3 i 4 łańcuchu.
Gdy tego nie zrobimy, otrzymamy: `'Ja mamtegopouszy'`

Należy zauważyć, że równoważne zapisy to:
`"Hello " + "World"` czy `"Hello" + " World"` lub `"Hello" + "" + "World"`

Formatowanie łańcucha znaków.

Omówienie tylko w formie zasygnalizowania
W Python 2.6 wprowadzono metodę **`str.format()`** o nieco innej składni z istniejącym operatorem %.
Formatowanie jest jedną z wielu możliwości połączenia łańcuchów znaków.
W Pythonie możemy formatować wartości wewnątrz łańcucha znaków.
Wewnątrz łańcucha tekstowego możesz wstawiać specjalne znaki formatujące.
Do wyboru: **`%s`** (dla łańcuchów), **`%c`** (dla pojedynczych znaków), **`%d`** (dla liczb całkowitych dziesiętnych), **`%f`** (liczby rzeczywiste), `%.` - Liczba zmiennoprzecinkowa ze stałą liczbą cyfr po kropce.
Następnie podajemy listę (a właściwie krotkę), zawierającą odpowiednie zmienne. Jest to powszechnie stosowana konwencja formatowania tekstu w Pythonie i warto ją wykorzystywać, ponieważ jest przejrzysta i wygodna.
Uwaga. Instrukcja print po wypisaniu tekstu automatycznie wstawia znak nowej linii. Jeśli chcemy tego uniknąć - po jednej instrukcji print, a przed kolejną wstawiamy znak przecinka.
Chociaż możemy tworzyć bardzo skomplikowane wyrażenia, jednak najczęściej w prostych przypadkach wystarczy wykorzystać **`%s`** (Napis) lub **`%d`**, aby wstawić pewien łańcuch znaków wewnątrz innego.
Po cudzysłowie zamykającym napis umieszczamy operator formatowania **%**, a po nim zmienne lub wyrażenia odpowiadające użytym wcześniej symbolom. Zmienne muszą być umieszczone w nawiasach () – Krotka - i są zapisane jako szereg oddzielonych przecinkami wartości, chyba że jest tylko jedna zmienna. Wtedy możemy je pominąć.
W trywialnym przypadku formatowanie daje ten sam wynik co łączenie.

Przykład 1 – formatowanie łańcucha:

```
>>> #!/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.
>>> |
```

`% operator (modulo); '%', oznacza początek definicji.`

Przykład 2:
Zestaw danych, które mają być *wypisane*, możemy zapisać w swego rodzaju uporządkowanej tablicy, którą nazywamy krotką (ang. nazwa to "tuple").

```
>>> #!/usr/bin/env python
>>> # -*- coding: utf-8 -*-
>>> # Tworzymy 'krotkę' - dane
>>> dane = ("Zbyma", 3, 42)
>>> print "%s mieszka w bloku %d w mieszkaniu %d" % dane
Zbyma mieszka w bloku 3 w mieszkaniu 42
>>> |
```

Przykład 3 - ***Formatowanie liczb*****:**

```
>>> #!/usr/bin/env python
>>> # -*- coding: utf-8 -*-
>>> print "Wczorajsza cena dolara: %f" % 3.5984
Wczorajsza cena dolara: 3.598400
>>> print "Dzisiejsza cena dolara: %.2f" % 3.6524
Dzisiejsza cena dolara: 3.65
>>> print "Zmiana w stosunku do dnia wczorajszego: %+.2f" % 1.5
Zmiana w stosunku do dnia wczorajszego: +1.50
>>> |
```

Pole formatowania **`%f`** traktuje wartość jako liczbę rzeczywistą i pokazuje ją z dokładnością do 6 miejsc po przecinku.
Modyfikator "**.2**" pola %f umożliwia pokazywanie wartości rzeczywistej z dokładnością do dwóch miejsc po przecinku.
Można nawet łączyć modyfikatory. Dodanie modyfikatora + pokazuje plus albo minus przed wartością, w zależności od tego jaki znak ma liczba. Modyfikator "**.2**" został na swoim miejscu i nadal nakazuje wyświetlanie liczby z dokładnością dwóch miejsc po przecinku.

Wcięcia

http://python.edu.pl/byteofpython/2.x/05.html#wciecia
Funkcje w Pythonie nie posiadają sprecyzowanych początków i końców oraz żadnych nawiasów służących do zaznaczania, gdzie funkcja się zaczyna, a gdzie kończy.
Jedynym separatorem jest dwukropek **(:)** i **wcięcia kodu**.
Białe znaki w Pythonie na początku linii są znaczące. Nazywamy je **wcięciami**. Wiodące białe znaki (mamy cały czas na myśli spacje i znaki tabulacji) na początku linii logicznej są brane pod uwagę przy określaniu stopnia wcięcia danej linii logicznej, co z kolei pozwala Pythonowi grupować polecenia.
Łatwo się domyślić, że polecenia, które są tak samo ważne **muszą** mieć takie samo wcięcie.
Każdy taki zestaw poleceń nazywamy **blokiem**.
Musimy również zapamiętać, że nieprawidłowe wcięcia pociągają za sobą czasem trudne do znalezienia błędy. Nigdy **nie mieszamy Tab-ów i spacji** w plikach Pythona!. ***Tab*** *najlepiej wyłączyć z użycia*, natomiast *cztery spacje* dla kodu GIMP, są standardem używanym we wtyczkach GIMP-a.
Przykład:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Nazwa pliku: biale_znaki.py
i = 5 # polecenie, ponieważ zleca Pythonowi wykonanie czegoś: w tym przypadku łączymy nazwę zmiennej i z wartością 5.
 print 'Wartość zmiennej to ', i
```

Klikając **`Enter`**, otrzymamy następujący błąd:

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> #!/usr/bin/env python
>>> # -*- coding: utf-8 -*-
>>> # Nazwa pliku: biale_znaki.py
>>>
>>> i = 5
>>>  print 'Wartość zmiennej to ', i
  File "<input>", line 1
    print 'Wartość zmiennej to ', i
   ^
IndentationError: unexpected indent
>>> |
```

Moduł **pdb** definiuje interaktywny debugger dla kodu źródłowego programów Pythona.
^ Wskazana przez Pythona spacja na początku linii **miejsce błędu**..
Oraz pojawił się opis:
IndentationError: unexpected indent
Błąd wcięcia: Nieoczekiwane wcięcie (widać spację na początku linii przed print)
Błąd wychwycony przez Pythona oznacza, że składnia tego programu jest nieprawidłowa, czyli został źle napisany
Powtarzamy:

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> #!/usr/bin/env python
>>> # -*- coding: utf-8 -*-
>>> # Nazwa pliku: biale_znaki.py
>>>
>>> i = 5
>>> print 'Wartość zmiennej to ', i
Wartość zmiennej to  5
>>> |
```

Jak widać, wypisywanie danych (na standardowe wyjście) jest bardzo proste, wystarczy użyć słowa kluczowego **print**.
Wyrażenie **print** może przyjąć każdy typ danych, na przykład łańcuchy znaków, liczby całkowite i inne wbudowane typy danych jak słowniki i listy.

[print jest *instrukcją (w nowym Pythonie **3.x** jest to już funkcja i używa się jej nieco inaczej, ale tym nie będziemy się teraz martwić.* Możemy jej przekazać jeszcze jeden argument, mówiący o tym co dokleić do wyświetlanej treści. Stąd powinniśmy posłużyć się funkcją print, z zaznaczeniem, że na końcu treści nie chcemy wyświetlać nic (łącznie ze znakiem nowego wiersza): print(i, end="")**]**

Wniosek:
Nie możemy dowolnie zaczynać nowych bloków poleceń (za wyjątkiem oczywiście bloku głównego, który kiedyś musimy przecież rozpocząć).

Jak używać wcięć?
Powtażam: Nie używamy mieszaniny tabulatorów i spacji do stosowania wcięć. *Zalecane jest*, używanie *pojedynczego tabulatora* lub *czterech spacji* na każdy jeden stopień wcięcia.
Wybieramy jeden z powyższych sposobów stosowania wcięć i **stale** używamy *tylko* tego sposobu.

http://level1wiki.wikidot.com/syntax-error
SyntaxError: invalid syntax

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> Stala = "Co to jest"
>>> Stala = "Co to jest?
  File "<input>", line 1
    Stala = "Co to jest?
                       ^
SyntaxError: EOL while scanning string literal
>>> |
```

W drugim przypadku dodano **?**, a zapomniano **"**.

```
>>> "Hallo" + 1
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> |
```

TypeError: Nie można łączyć 'str i obiektów 'int'
Ale jeśli określimy funkcję str(1) to "Hallo" + str(1) => 'Hallo 1'

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
>>> print(10*abc)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'abc' is not defined
>>> print (10*'abc')
abcabcabcabcabcabcabcabcabcabc
>>> |
```

Przecinek

Przy pomocy **`print`** możemy, drukować na ekran różne wartości w jednej linii.
W tym celu podajemy ciąg wartości, które chcemy wyświetlić, oddzielając je przecinkiem.
Każda wartość jest wtedy wyświetlana w tej samej linii i oddzielona spacją (znak przecinka nie jest drukowany).

```
>>> print "a","b","c"
a b c
>>> |
```

`Print` nie tylko oddziela argumenty spacją, ale dodaje na końcu znak przejścia do nowej linii (co widać powyżej).
Jeżeli chcemy wypisać dwie wartości (literały) bez odstępu musimy je jawnie skleić, czyli napisać jeden obok drugiego bez żadnego operatora.
Co zrobić, jeżeli nie chcemy znaku przejścia do nowej linii? Magiczny przecinek przybywa na ratunek:

```
>>> print "a","b","c",
a b c
```

Dzięki temu przecinkowi Python nie przechodzi do następnej linii napisu, lecz przesuwa tylko kursor o jedną pozycję w prawo (zostanie wówczas wypisana spacja). Różnicę można zobaczyć tylko podczas wykonywania programów.
.

```
print'Witaj w świecie Pythona,', 'czeka Cię wspaniała przygoda.', 'Więc na co jeszcze czekasz?'
```

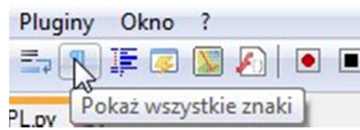
Wspaniale a przynajmniej takie się wydaje do momentu, gdy kilka razy o tym przecinku zapomnimy**…**

Znaki końca linii to **\r** (*return*) oraz **\n** (*new line*), które mogą pojawiać się pojedynczo lub razem.
W zależności od systemu operacyjnego ich użycie jest inne, a wyróżnić można trzy rodzaje:

- **CR** – wykorzystuje tylko **\r**, skrót pochodzi od anglojęzycznego zwrotu *carriage return* (*powrót karetki*) - MAC
- **LF** – wykorzystuje tylko \n, skrót pochodzi od anglojęzycznego zwrotu *line feed* (w wolnym tłumaczeniu: *nowa linia*) - UNIX
- **CR|LF** – połączenie obu powyższych, wykorzystuje \r\n - Windows

```
43 register(LF
44     "python_fu_hello_world",LF
45     "Obraz Hello world ",LF
46     "Tworzymy nowy obraz z Naszego łańcucha tekstu",LF
47     "Akkana Peck",LF
48     "Akkana Peck",LF
49     "2010",LF
50     "Hello world (Py)...",LF
51     "",     # Tworzenie nowego obrazu, nie działa na istniejącymLF
52     [LF
53         (PF_STRING, "string", "Napis", 'Hello, world!'),LF
54         (PF_FONT, "font", "Wybór czcionki", "Sans"),LF
55         (PF_SPINNER, "size", "Rozmiar czcionki", 50, (1, 3000, 1)),LF
56         (PF_COLOR, "color", "Kolor tekstu", (1.0, 0.0, 0.0))LF
57     ],LF
58     [],LF
59     hello_world, menu="<Image>/File/Create")LF
60 LF
61 main()LF
62
```

Część okna **Notepad++** z przykładową przetłumaczoną wtyczką, ze zmienioną skórką.
[Powiększanie/zmniejszanie podglądu w Notepad++ tłumaczonej wtyczki Ctrl + Kółko myszki, podświetlanie składni, możemy łatwo znaleźć sparowanie nawiasów, podświetla wcięcia, nawiasy i klamry **]**



Po kliknięciu przycisku jak widać powyżej wyświetlą się w oknie Notepad++ znaki *LF nowa linia.*

Jak widać, Notepad++ ma wyświetlanie znaków niedrukowanych, rozpoznawanie trybu LF/CRLF, możliwość ustawienia domyślnego dla nowych plików.
Jak uzyskać inny znak np. "CR LF"?.
Menu => Edycja => Konwersja znaku końca lini => Konwertuj na format Windows lub UNIX lub MAC.
Więcej o wykorzystaniu Notepad++ poniżej.

Funkcja w Pythonie

Definicja funkcji w Pythonie polega na użyciu słowa kluczowego **def**, podaniu nazwy funkcji i w nawiasach okrągłych ewentualnej listy argumentów. Funkcje w Pythonie nie posiadają sprecyzowanych początków i końców oraz żadnych nawiasów służących do zaznaczania, gdzie funkcja się zaczyna, a gdzie kończy. Definicję kończymy znakiem dwukropka (**:**), po którym wpisujemy w następnych liniach, pamiętając o wcięciach, ciała funkcji.
Funkcja może, ale nie musi zwracać wartości. Jeżeli chcemy zwrócić jakąś wartość używamy polecenia **return**. **return** - zwraca podane wyrażenie (zmienna, obiekt itp.). Jeżeli chcemy zwrócić więcej niż jedną zmienną to stosujemy tuplę (zmienna, zmienna, zmienna…). Zmienne w definicji funkcji mogą mieć przypisane wartości domyślne (jeżeli w wywołaniu funkcji nie podamy wartości parametru to użyta zostanie wartość domyślna). Zmienne utworzone wewnątrz funkcji nie są dostępne poza nią, lecz można je zdefiniowiać jako zmienne globalne za pomocą operatora **global**.

Instrukcje warunkowe...

Instrukcja warunkowa...(*jeżeli*) jest używana do sprawdzania warunków i nazywa się "**if**", a składnia jest taka:

>>> if warunek: #zrób coś

```
>>> x=2
>>> y=2
>>> if x==y:
... print ('x i y są równe')
  File "<input>", line 2
    print ('x i y są równe')
        ^
IndentationError: expected an indented block
>>> if x==y:
...     print ('x i y są równe')
...
x i y są równe
```

Jak widać najpierw pojawił się komunikat o błędzie wcięcia, o czym zapomniałem!
Blok kodu *wewnątrz* klauzuli **if** i **else** musi być wcięty więcej niż gdyby była sama if lub **else** przykł.:

```
␣␣␣␣if radius < 1:
    ␣␣radius = 1 (tutaj znaki ␣ - to symbole graf. oznaczające spację )
```

Znak **równości** ("**=**") jest używany do przypisania wartości do zmiennej. Przypisanie do zmiennej nie jest wypisywane przez interpreter. (**Elif** jest skrótem **else** i **if**.)

Operatory porównywania:

Natomiast znak (**==** równe co do wartości), jest używany jeżeli coś **równo, równa się** drugiemu np. **x==y**.
Znak (**!=**) – negacja powyższego, czyli Nie jest równe.

operator	znaczenie
<	mniejsze
<=	mniejsze lub równe
>	większe
>=	większe lub równe

Przyrównania (==, <=, <=, >=, <, >, !=) w zasadzie są łączne lewostronnie, ale w praktyce w przypadku zwykłych obiektów wynik nie zależy od kolejności wykonania.

Znak **--** jest napisem o długości jeden.

```
>>> if 1 = 2:
         ^
SyntaxError: invalid syntax
```

Python informuje nas o *błędzie składni* (ang. *syntax*), co oznacza że nasz skrypt nie jest poprawnie zbudowanym plikiem Pythona. Kiedy jesteśmy świadkami błędu wykonywania skryptu Pythona, interpreter podaje nam dokładne miejsce, gdzie wystąpił błąd. *Debugowanie* programu zaczynamy właśnie od tej linii.

Teraz do Konsoli wpisujemy:
`>>> greeting = "hello" + " world`
Klikamy Enter i wyświetli się:

```
>>> greeting = "hello" + " world
  File "<input>", line 1
    greeting = "hello" + " world
                               ^
SyntaxError: EOL while scanning string literal
>>> |
```

End-of-line (EOL)
Koniec linii **(ang. *end of line, EOL*) –** znak lub sekwencja znaków oznaczająca zakończenie linii tekstu.
Programista (amator lub zawodowiec), *musi* nauczyć się czytać
komunikaty o błędach i zebrać jak najwięcej informacji, jak to możliwe.
W tym przypadku, Błąd mówi, że linia nie może być wykonana, ponieważ to
nie kompilacja: mamy SyntaxError.
SyntaxErrors pokazuje, gdzie występuje błąd: widzimy strzałkę w górę ^
w ramach linii kodu. Wskazuje na koniec linii.
W połączeniu z komunikatem o błędzie End-of-line (EOL)
"EOL while scanning string literal" => EOL podczas skanowania łańcucha znaków
przyczyna błędu jest rozwiązana:
Pythonowi zabrakło końca linii ciągu łańcucha znaków, lub np. jak powyżej – po
słowie "world *brak cytowania*.
(W Goggle szukamy co oznacza, błąd końca linii.) http://pl.wikipedia.org/wiki/Koniec_linii

Funkcja **`len ()`**– zwraca długość (liczbę elementów) obiektu. Argumentem może być ciąg (string, krotki lub listy) lub odwzorowania (słownika).

```
>>> len("Ahoj")
4
>>> a = "Ahoj"
>>> b = "swiat"
>>> len((a+b)*2)
18
>>> |
```

Identyfikatory

Zmienne są przykładami identyfikatorów. *Identyfikatory* to nazwy, które nadajemy *czemuś* do zidentyfikowania tego *czegoś*.
Tworząc je trzymamy się poniższych zasad:

- Pierwszym znakiem identyfikatora musi być mała lub duża litera alfabetu łacińskiego, (polskie litery diakrytyczne *są niedopuszczalne*) albo podkreślnik (_).
- Pozostałe znaki mogą zawierać małe lub duże litery alfabetu łacińskiego, podkreślniki oraz cyfry (0 – 9).
- Wielkość znaków w identyfikatorze jest ważna.

http://www.tutorialspoint.com/python/python_if_else.htm http://learnpythonthehardway.org/book/ex30.html
https://pl.python.org/docs/tut/node5.html

Komentarze

W Pythonie istnieją dwa rodzaje komentarzy:

- *komentarz wierszowy* rozciąga się od znaku **#** aż do końca wiersza włącznie
  `instrukcja` **`# tekst aż do końca bieżącego wiersza jest komentarzem`**
- *komentarz blokowy* obejmuje dowolny tekst ujęty w potrójne cudzysłowy
- **`""" komentarzem jest tekst`**
- **`    znajdujący się pomiędzy ogranicznikami`**
- **`    o postaci potrójnych cudzysłowów technicznych.`**
- **`"""`**

```
'''  zamiast cudzysłowów można także użyć
     ograniczników o postaci potrójnych apostrofów.
'''
```

Wiele przykładów w tym poradniku, nawet te wprowadzone w linii poleceń, zawierają komentarze.
Jak już podano, w Pythonie komentarz rozpoczyna się od znaku `"#"` «**hash**» (a nie średnika) i ciągnie się w prawo, aż do końca fizycznego wiersza. Po napotkaniu # Python ignoruje wszystkie pozostałe znaki w danej linii (więc można pisać, co się chce). Jest użyteczne przede wszystkim jako notki pozostawione dla przyszłego czytelnika programu. Komentarze są bardzo pomocne.
Mogą się one okazać bardzo przydatne dla osób, które będą czytały Nasz program. Również Nam z pewnością się przydadzą, kiedy będziemy czytać swój program po długim czasie od napisania.
Powinny one wyjaśniać, *co* program wykonuje, albo też informować o rzeczach, które czekają na poprawienie/dokończenie/zrobienie.
Dwa wyjątki:
Komentarz może pojawić się na początku linii lub kończyć instrukcję, lecz nie może być zawarty w literale - ciągu znaków.
Znak «**hash**» ("**#**") w ciągu znaków jest po prostu zwykłym znakiem, jednym z wielu tego ciągu.
O dalszych dwóch specjalnych wyjątkach będzie poniżej.
Komentarze są czymś, co pomoże Nam zrozumieć skrypt, ale nie mają one wpływu na faktyczną realizację. Innymi słowy, komentarze nie są w ogóle oprogramowaniem, jest to czysty tekst.

Trochę więcej na temat modułów Pythona

Prawdziwa siła Python leży w jego modułach bibliotecznych. Dostarczają one duży i spójny zbiór interfejsów programowania aplikacji (Application Programming Interfaces - API) dla wielu bibliotek systemowych (często w sposób niezależny od systemu operacyjnego).
Python pozwala umieszczać pożyteczne definicje obiektów oraz instrukcje w osobnym pliku zwanym modułem (bibliotecznym). Możemy importować definicje z danego modułu do innych modułów lub do modułu głównego, czy trybu interaktywnego. Jeżeli nazwa modułu to ***abc***, to nazwa pliku modułu ma postać ***abc.py***.
Ilekroć możliwe, należy korzystać z funkcji oferowanych przez te moduły do manipulacji plików, katalogów oraz ścieżek. Moduły te są *opakowaniami* dla specyficznych modułów platformy, a więc specyficznych funkcji, które pracują w systemie UNIX, Windows, Mac OS i innych platform obsługiwanych przez Pythona.
Zawsze, gdy chcemy wykorzystać kod stworzony przez kogoś innego (lub przez nas samych), z pomocą przychodzą **moduły**. Możemy stworzyć bibliotekę z funkcjami, które często wykorzystujemy, a następnie używać jej w kolejnych programach. Tak samo możemy używać bibliotek innych osób.
W Pythonie mamy dwa sposoby importowania modułów. Obydwa są przydatne.

Jednym ze sposobów jest użycie polecenia **`import`** ***`nazwa_modułu`***.
Pozwala to użytkownikowi korzystać z obiektów zadeklarowanych w module.
W tym sposobie wczytywania modułu zdefiniowane w nim obiekty noszą nazwy postaci:
`nazwa_modułu.nazwa_obiektu`.
(Niektóre moduły, tak jak **`sys`** są **wbudowane** w samego Pythona. Wbudowane moduły zachowują się w ten sam sposób co pozostałe, ale nie mamy bezpośredniego dostępu do ich kodu źródłowego, ponieważ tak jak wymieniony moduł **`sys`** są napisane w C).)
Pod adresem http://docs.python.org/library/index.html znajduje się opis modułów dostępnych standardowo w Pythonie, których nie trzeba instalować – wystarczy tylko zaimportować. Jest to bogata biblioteka, z którą warto się zapoznać, aczkolwiek większość z dostępnych modułów i tak poznaje się dopiero z czasem.
Najpierw moduł jest ładowany w wykonujący się skrypt Pythona. Potem jego zawartość jest odczytywana tak, jakbyśmy to sami napisali ten kod wcześniej. Bardzo często jeden moduł importuje inne.
Jeśli zdarzy się, że interpreter dwa razy napotka polecenie dołączenia tego samego modułu, to drugie polecenie zostanie zignorowane.
Przykłady:
Aby użyć zawartości modułu, należy go ***zaimportować***. Dotyczy to również modułów ze standardowej biblioteki Pythona, od czego właśnie zaczynamy.
`import os` - służy przede wszystkim do komunikacji z systemem operacyjnym, dostarcza interfejs do API systemu operacyjnego. Moduł **`os`** zawiera dwa submoduły **`os.sys`** (tak samo jak **`sys`**) i **`os.path`**, które są przeznaczone odpowiednio do systemu i katalogów. http://www.thomas-cokelaer.info/tutorials/python/module_os.html
http://www.thomas-cokelaer.info/tutorials/python/basics.html#basics-module
http://www.thomas-cokelaer.info/tutorials/python/module.html#docmodule

```
>>> import os
>>> a = os.path.join('path', 'sub_path')
>>> print a
path\sub_path
>>> |
```

Polecenie **os.path.join** jest bardzo użyteczne – pozwala na łączenie ścieżek w sposób niezależny od systemu operacyjnego
Do zmiennych środowiskowych systemu można uzyskać dostęp poprzez:
>>> **print os.environ** -- który jest katalogiem tylko do odczytu.

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> import os
>>> print os.listdir('.')
['bz2-1.dll', 'bzip2.exe', 'freebl3.dll', 'gdb.exe', 'gi
mp-2.8.exe', 'gimp-console-2.8.exe', 'gimp-console.exe
', 'gimp.exe', 'gimptool-2.0.exe', 'gimp_2_8_en.chm', 'g
imp_2_8_fr.chm', 'gimp_2_8_fr.chw', 'gspawn-win64-helper
-console.exe', 'gspawn-win64-helper.exe', 'icudata53.dll
', 'icui18n53.dll', 'icuio53.dll', 'icule53.dll', 'iculx
```

Powyższy kod prezentuje, jak najprościej można użyć biblioteki. W przykładzie zaimportowaliśmy *bibliotekę* "**os**", która służy przede wszystkim do komunikacji z systemem operacyjnym. Biblioteka ta zawiera całą gamę funkcji, my jednak użyliśmy tylko jednej: **'listdir()'**, która zwraca listę plików i katalogów znajdujących się we wskazanym katalogu (w przykładzie jest to kropka '.', czyli katalog bieżący). Należy zwrócić uwagę, że *nazwę funkcji poprzedziliśmy nazwą biblioteki*, z której ona pochodzi, oddzielając je od siebie kropką. Dzięki takiemu postępowaniu unikamy konfliktu nazw.
Po zaimportowaniu biblioteki jak widać, mamy dostęp do wszystkich funkcji, które ona udostępnia.

import sys - moduł **sys** zawiera **narzędzia** związane z systemem, czyli po prostu mówimy Pythonowi, że chcemy go używać. Moduł **sys** zawiera polecenia związane z Pythonem i jego środowiskiem, czyli *sys*temem. Zawiera listę ścieżek na których Python wyszukuje moduły. Moduł **sys** zawiera wiele zmiennych i funkcji, ale ta, która jest najbardziej dla nas przydatna, to **sys.path**. Gdy próbujemy zaimportować moduł, Python odszukuje go we wszystkich folderach wykazanych przez **sys.path**. Jeśli instalujemy moduł w jakimś miejscu i chcemy, aby Python go znalazł, to musimy dołączyć tą lokalizację do **sys.path**.

Przykładowe inne moduły:
gettext.install("gimp20-python", gimp.locale_directory, unicode=True)
- instaluje **gettext** pliki wielojęzycznych tłumaczeń przygotowane przez autora, aby wtyczka mogła wyświetlać etykiety nie tylko w języku ang. ale również w języku lokalnym **_()** (szczegóły dalej)
import re wyrażenia regularne, pozwalają na odnalezienie znaków pasujących do podanego wzorca
import urllib - bardzo przydatna biblioteka do ściągania stron WWW. Ale nie tylko.
import math – jest ważnym modułem, wchodzący w skład każdej instalacji Pythona. Zawiera on deklaracje wielu przydatnych funkcji matematycznych. Musimy odwoływać się do nich za pomocą notacji moduł.funkcja, np.: **math.sqrt()** – zwraca pierwiastek kwadratowy.
Przykład zastosowania:

```
>>> import math
>>> math.sqrt(5.0)
2.23606797749979
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/2)
1.0
```

Moduły są dla programu takimi samymi obiektami, jak wszystkie inne (np. zmienne). Możemy się o tym przekonać, sprawdzając typ wczytanego modułu albo listując jego zawartość:

```
>>> import math
>>> type(math)
<type 'module'>
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh
', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', '
copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc
', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod
', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan
', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf',
'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', '
tanh', 'trunc']
>>> |
```

Dla usunięcia zaimportowanego modułu z pamięci programu wystarczy użyć instrukcji ***del***, np. **`del math`**
Operacja taka nie czyni żadnej krzywdy plikowi, z którego moduł został załadowany.
Python jest jednak o wiele bardziej elastyczny, jeśli chodzi o importowanie.
Istnieje inny sposób, który realizuje tą samą czynność, ale posiada pewne różnice.
Ten inny sposób wczytywania modułów, oparty jest na wariancie składni polecenia **`import`** z wykorzystaniem słów kluczowych **`from`** oraz **`as`**, co umożliwia dostęp do obiektów zdefiniowanych w module bez poprzedzania ich *nazw nazwą_modułu*, lub z poprzedzeniem ich dowolnie ustaloną nazwą.

`import, from i as` są słowami kluczowymi Pythona.

Drugi sposób najczęściej wykorzystuje instrukcję ***`from module import`*:**
Po co, importować (jak powyżej) wszystkie funkcje z modułu (biblioteki) "**`os`**", jeśli chcemy użyć tylko jednej.
Najpierw importujemy funkcję z modułu **`os`**, a następnie ją wywołujemy np:

`from os import listdir.`

Atrybuty i metody danego modułu są importowane bezpośrednio do lokalnej przestrzeni nazw, a więc będą dostępne bezpośrednio i nie musimy określać, z którego modułu korzystamy.
Różnice są dwie: pierwsza polega na tym, że importujemy wyłącznie tę jedną funkcję, druga - nie poprzedzamy nazwy funkcji *nazwą_modułu*.
Aby zaimportować kilka różnych funkcji, po prostu podajemy ich nazwy po przecinku.
Trzeba jednak pamiętać, *nazwy muszą być unikalne*. Jeśli utworzyliśmy wcześniej obiekt o takiej nazwie, jak obiekt, który chcemy zaimportować, lub też w dwóch różnych modułach (bibliotekach) występują obiekty o takich samych nazwach, wystąpił by konflikt. Możemy go uniknąć, ustalając nazwę, pod jaką chcemy używać danego obiektu (te reguły odnoszą się do funkcji, klas i zmiennych, ponieważ w Pythonie wszystko jest obiektem, traktujemy je na równi).
Możemy więc importować określone pozycje albo skorzystać z **`from module import *`**, aby zaimportować wszystko.
Jest ona jednak niezalecana, gdyż powoduje zaśmiecanie lokalnej przestrzeni nazw.

W Naszych wtyczkach stosujemy:
`from gimpfu import *` `Takie wywołanie sprawi, że zaimportowane zostaną wszystkie obiekty dostępne w module` **`gimpfu`**`, przy czym ich nazw w wywołaniach nie będzie trzeba poprzedzać nazwą modułu.`
Jeśli nie jest potencjalna możliwość wykorzystania większości obiektów, zalecane jest wymienienie ich w liście obiektów do importu (**`from .. import ..`**) lub import całego modułu do osobnej przestrzeni nazw (**`import ..`**).

Odszukanie i zmiana CWD
Lokalizacja, w której znajduje się użytkownik w danej chwili nawigowania w środowisku, określana jest jako *katalog bieżący* (do którego mamy bezpośredni dostęp).
Jeśli chcemy dowiedzieć się, który katalog jest katalogiem *bieżącym*, można użyć funkcji o nazwie **`getcwd()`** ("**get c**urrent **w**orking **d**irectory") która jest w module os:

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> import os
>>> os.getcwd()
'C:\\Users\\ZbigniewMalach\\Desktop\\GIMPPortab\\gimp_2_
8_15-64bits_20140901_Portable\\bin'
>>> |
```

Zwraca bieżący katalog roboczy (Zauważmy, że Python w ścieżce Windows używa znaków cytowania.).

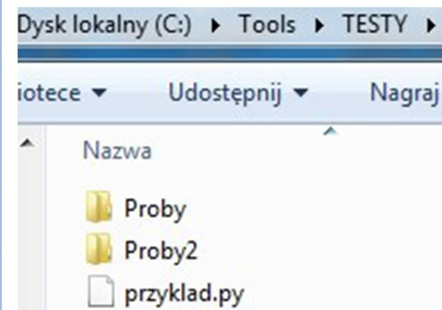
ale

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> import os
>>> print(os.getcwd())
C:\Users\ZbigniewMalach\Desktop\GIMPPortab\gimp_2_8_15-6
4bits_20140901_Portable\bin
>>> |
```

(Zauważymy, że instrukcja **print**, może zostać użyta do wypisywania ciągów znaków bez znaków cytowania (znak ucieczki) - nie używa podwójnych ukośników.)

Aby zmienić bieżący katalog dyskowy na inny używamy funkcji **chdir()**(' **ch**ange **dir**ectory'), którą także zawiera moduł **os**. Funkcji **chdir()** podajemy jeden argument - napis nazwa (ścieżka względna lub bezwzględna) pliku do katalogu, na który chcemy zmienić aktualny, możemy sprawdzić jego zawartość:

```
>>> #!/usr/bin/env python
>>> # -*- coding: utf-8 -*-
>>> import os
>>> os.getcwd() # Sprawdzenie, który katalog jest katalogiem
bieżącym
'C:\\Users\\ZbigniewMalach\\Desktop\\GIMPPortab\\gimp_2_8_15-
64bits_20140901_Portable\\bin'
>>> os.chdir('C:\\Tools') # Zmieniamy bieżący katalog dyskow
y na inny
>>> os.getcwd() # sprawdzamy czy jest to katalog bieżący
'C:\\Tools'
>>> os.listdir('.') # Wylistujemy co zawiera ten katalog
['LabCurves', 'TESTY']
>>> os.listdir('TESTY') # Stwierdzamy co zawiera katalog TEST
Y
['przyklad.py']
>>> print('przyklad')
przyklad
>>> os.mkdir(r'C:\Tools\TESTY\Proby') # Tworzymy kolejny kata
log o nazwie Proby wewnątrz TESTY stosując inny format ścieżk
i bezwzględnej
>>> os.chdir('C:\\Tools\\TESTY') # Zmieniamy bieżący katalog
dyskowy Tools na TESTY
>>> os.getcwd() # sprawdzamy czy TESTY jest teraz katalogiem
bieżącym
'C:\\Tools\\TESTY'
>>> os.mkdir('./Proby2') # można również utworzyć nowy katalo
g, w ramach TESTY używając względnej nazwy ścieżki
>>> |
```



Oczywiście, równie dobrze możemy podać inny format ścieżki absolutnej do katalogu:

```
>>> os.listdir(r'C:\Tools')
```

lub zastosować:

```
os.listdir() # wywołanie z użyciem namespace
from os import listdir # import obiektu listdir (funkcja) do lokalnej
przestrzeni nazw
listdir() # wywołanie funkcji z lokalnej przestrzeni nazw, nie musimy
poprzedzać nazwy funkcji nazwą_modułu.
```

Kolejny katalog tworzymy za pomocą funkcji **os.mkdir(), Przykładowo:**

```
>>> os.mkdir('./python') # utworzymy nowy podkatalog python, w ramach
bieżącego katalogu roboczego CWD, tutaj TESTY, używając względnej nazwy ścieżki.
Będziemy mogli w nim zapisywać Nasze wtyczki.
>>> os.chdir('../')#tym możemy wrócić do katalogu nadrzędnego z katalogu TESTY
```

Komentarz:

Odwoływanie się do pliku w systemie

- Pliki zorganizowane są w hierarchiczną strukturę nazywaną drzewem katalogowym (ang. **directory tree**). Na szczycie drzewa znajduje się katalog nazywany katalogiem głównym (ang. **root directory**), W systemach UNIX oznaczany symbolem ukośnika (/), a w systemach DOS przez ukośnik odwrotny (\). W systemach uniksowych katalog główny jest jeden, w systemach DOS może być ich kilka (tyle, ile partycji). Nazwa pliku musi być unikatowa tylko w obrębie katalogu.

- Pełna nazwa pliku odzwierciedla strukturę drzewa i określana jest jako *Ścieżka bezwzględna* (ang. **Absolute path name**). Podaje ona sekwencję katalogów prowadzących do pliku czyli określa bezwzględne położenie pliku w drzewie katalogowym, np.: ***C:\Program Files\Zbyma*** oznacza podkatalog *Zbyma* w katalogu *Program Files* na dysku C – niezależnie od tego, gdzie się teraz znajdujemy. Czyli bezwzględna ścieżka to cały adres do pliku.
- Bieżący Katalog Roboczy **(CWD** - **C**urrent **W**orking **D**irectory**)** jest wykorzystywany do tworzenia nazw względnych (ang. **Relative path name**). *Ścieżka względna* podaje położenie pliku względem bieżącego katalogu roboczego, czyli gdy program i plik z danymi znajdują się w tym samym miejscu. Jeśli odniesienie do Naszego pliku nie startuje od góry **(np. 'TESTY/Proby2'),** Python zakłada, że startuje w **bieżącym katalogu roboczym** ("**relative path**"). Gdy użyjemy ścieżkę w stylu Linux **/** (a nie litery dysku, mimo że jesteśmy w systemie Windows), Python usiłuje pracować pomimo różnic między systemami operacyjnymi.

W systemie **Linux** - Ścieżki dostępu bezwzględne zaczynają się zawsze od nazwy katalogu głównego "/" ścieżki względne nigdy nie zaczynają się od nazwy katalogu głównego i identyfikują plik/katalog z punktu widzenia katalogu bieżącego.
Nazwy katalogów w ścieżkach oddzielamy znakiem "/", a więc odwrotnie niż w Windows (DOS).
`"."` oznacza katalog bieżący,
`".."` katalog nadrzędny.

W systemie **Windows**, istnieje kilka dodatkowych sposobów odwoływania się do pliku. To dlatego, że natywnie, plik ścieżki Windows wykorzystuje ukośnik odwrotny (**backslash**) "****" zamiast ukośnika "**/**" (**slashes**).
Python pozwala na zastosowanie, ale istnieje kilka pułapek, na które trzeba uważać.
Podsumujmy je:

- Python pozwala używać stylu ukośnika OS X / Linux (**slashes**) "**/**", nawet w systemie Windows. W związku z tym, można odwołać się do pliku jako **`'C:/Users/Zbyma/Desktop/abc.py'`.**
- W przypadku korzystania z odwrotnego ukośnika (backslash) **"\"**, należy pamiętać aby zastosować ucieczkę **(escape)** do każdego wystąpienia, ponieważ (backslash) **"\"** to znak specjalny w Pythonie:
  `'C:\\Users\\Zbyma\\Desktop\\abc.py'`
- Alternatywnie, można poprzedzić łańcuch całej nazwy pliku znacznikiem **"r"** (**rawstring marker)**: (**`r'C:\Users\Zbyma\Desktop\abc.py'`**). W ten sposób, każdy znak pojawiający się po odwróconym ukośniku jest pozostawiany bez zmian, *pozostawiane są również wszystkie odwrócone ukośniki* i nie trzeba stosować ucieczki do każdego odwrotnego ukośnika. Na przykład, literał napisowy r"\n" składa się z dwóch znaków: odwróconego ukośnika i małej litery "n"

Sposób informowania Pythona o kodowaniu znaków – dotyczy Python 2.x (!!) oraz tłumaczenie wtyczek.

Muszę poruszyć zagadnienie kodowania do wszystkich języków świata, zestawu znaków - **fontów**, który w zamierzeniu miał obejmować wszystkie pisma używane na świecie (aby nie zobaczyć krzaczków), czyli wielojęzycznych pluginów.

Jak podano powyżej, Python generalnie nie przejmuje się komentarzami oznaczanymi notacją:
specjalny znak **# hash =>** (kratka, krzyżyk, płotek)
Istnieją jednak dwa wyjątki:
Pierwszym, wyjątkiem jest sytuacja, kiedy pierwsze dwa znaki w pierwszej linii programu to **`#!`**
(tzw. *shebang*), po których bezpośrednio następuje ścieżka do programu.

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

Ścieżka ta jest informacją dla systemu, oznaczającą, że w przypadku *wykonywania* programu powinien on być uruchamiany za pomocą tego właśnie interpretera, czyli, że musi znaleźć lokalizację Pythona za pomocą programu **`env`** do którego została podana ścieżka, i dopiero wtedy użyć go jako interpretera.
Drugim wyjątkiem jest sytuacja, gdy na samym początku lub w drugiej linii pluginu dodajemy deklarację kod:

```
# -*- coding: {tu wstawiamy nazwę: utf-8, iso-8859-2 lub windows-1250} -*-
# -*- coding: utf-8 -*-
```

Jeśli chcemy skorzystać z innego kodowania znaków zamiast **`UTF-8`** oczywiście napiszemy coś innego.
Dodając polskie znaki z reguły korzystamy z kodowania **`UTF-8`** lub czasami ISO-8859-2,
Python musi wiedzieć, że Nasza wtyczka nie jest w **ANSI**. Kodowanie **ANSI** znajdziemy gdy np. korzystamy z wtyczek (dodatków) stworzonych w języku angielskim, których problem kodowania znaków dotyka w małym stopniu. (Jak nazwa wskazuje **ANSI** to skrót od American National Standards Institute, instytucji ustalającej normy techniczne obowiązujące w USA.)

Jeśli zamierzamy stosować symbole narodowe (nie znane w alfabecie łacińskim) należy plik odpowiednio zakodować. (https://www.youtube.com/watch?v=sqPTR_v4qFA Introduction to UTF-8 and Unicode)
[Wg.: http://pl.wikipedia.org/wiki/Font "**Font** (ang., z łac. *fons* – źródło) – zestaw czcionek o określonych cechach zapisany w postaci elektronicznej, w jednym pliku. Obecnie upowszechniły się **fonty** (TTF – Tru Type Font i OTF – Open Type Font) w wersji dwubajtowej, czyli w standardzie **Unicode**, co umożliwia zapisanie w jednym pliku (i jednym foncie) do **65536** znaków.
Stało się przez to możliwe wygodne stosowanie naraz nie tylko alfabetów wszystkich języków indoeuropejskich, ale nawet najważniejszych znaków języków Dalekiego Wschodu. Jednocześnie jest miejsce na wszelkie ligatury, kapitaliki, indeksy, znaki specjalne, piktogramy etc., ale najważniejszą rzeczą jest to, że w Unicode jest tylko jeden standard kodowania znaków – obowiązujący na wszystkich platformach (np. polskie „ą" ma odtąd ten sam numer kodu na wszystkich komputerach świata)."]

Wpisując do pliku wtyczki linię j/w ustawiamy kodowanie znaków **jednego pliku**, a nie całego programu (program może się składać z wielu plików, w oddzielnym katalogu).
W GIMP-ie *najczęściej* spotykamy wtyczki (pluginy) jednoplikowe (poza filtrami, kompilowanymi w "C").
Jeśli nie zdefiniujemy kodowania znaków i zaczniemy stosować polskie znaki, konsola Python-Fu natychmiast nas o tym uprzedzi.
Korzystając z Unikodu zapobiegamy wielu problemom,
Jeśli chcemy zapisać tekst w języku polskim, najlepiej jest użyć edytora z obsługą Unikodu.

System kodowania wykrywamy za pomocą **`chardet`.**
Stworzenie łańcucha znaków Unicode w Pythonie jest tak samo proste jak w przypadku zwykłego:

`unichr(378)`

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> print unichr(378)
ź
>>> |
```

Za pomocą funkcji ***`unichr`***, dowiedzieliśmy się jakiemu znakowi odpowiada dana liczba.
W tym przypadku liczbie **378** odpowiada polska litera "**ź**".

Python automatycznie zakoduje wpisywany napis unikodowy, aby został poprawnie wyświetlony na naszej Konsoli (i GIMP).

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> ord('a')
97
>>> chr(99)
'c'
>>> ord('%')
37
>>> chr(115)
's'
>>> |
```

1. Funkcja **`ord`** zwraca liczbę, która odpowiada danemu symbolowi. W tym przypadku literze "**a**" odpowiada liczba **97**.
2. Za pomocą funkcji **`chr`** dowiadujemy się, jaki znak odpowiada danej liczbie. Liczbie **99** odpowiada znak "**c**", a liczbie **115** znak "**s**"
3. Procent ("**%**") odpowiada liczbie **37**.

W tym miejscu jeszcze dalsze ważne uwagi:
Podstawową zasadą w tłumaczeniu kodu źródłowego wtyczki jest **nie ruszać słów kluczowych**.
Słowa kluczowe są rozpoznawane przez interpreter i muszą pozostać w dokładnie takiej postaci, w jakiej napisał je Autor (programista).

Również nazwy zmiennych, funkcji itp. Konstrukcji zdefiniowanych w programie należy pozostawiać bez zmian. Po pierwsze dlatego, że są one zawsze pisane w języku angielskim i jest to swego rodzaju przyjęty nieformalnie standard.

Nazwy w jakimkolwiek innym języku w kodzie programu będą wyglądać po prostu dziwnie. Z drugiej strony, jeśli zmienimy nazwę zmiennej lub funkcji w jednym miejscu, musimy znaleźć wszystkie jej wystąpienia w innych miejscach programu, co powoduje niepotrzebne komplikacje i bardzo łatwo przy tym o pomyłkę.

Tłumacząc: nie można tłumaczyć częściowo, np.:

menu="<Image>/Filters/Render" **na np.:** menu="<Image>/Filters/Renderowanie"

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> #!/usr/bin/env python
>>> # -*- coding: utf-8 -*-
>>>
>>> u = 'idzie wąż wąską dróżką'
>>> print(u)
idzie wąż wąską dróżką
>>> |
```

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> """Witaj w świecie Pythona"""
'Witaj w \xc5\x9bwiecie Pythona'
>>> print u"""Witaj w świecie Pythona"""
Witaj w Åwiecie Pythona
>>> # -*- coding: utf-8 -*-
>>> print """"Witaj w świecie Pythona"""
"Witaj w świecie Pythona
>>> |
```

Ilustracja braku lub zastosowania `# -*- coding: utf-8 -*-`

UTF8 jest tylko jednym ze sposobów kodowania Unicode w 8-bitowy strumień bajtów.
W UTF-8, znaki nie mają stałej długości. Mogą zawierać od 1 do 6 bajtów.
Plik UTF-8 czasem startuje z znacznikiem kolejności bajtów (**BOM** (ang. ***B****yte* ***O****rder* ***M****ark*, który jest "nagłówkiem" pliku, i który mówi programowi odczytującemu plik, jakie kodowanie zastosowano w pliku.).
Plik może być zakodowany w UTF-8 i nie posiadać BOM. Przy pisaniu wtyczek w przypadku UTF-8 nie jest on potrzebny (kodowanie w UTF-8 jest jednoznaczne) a wręcz przeciwnie - powoduje częste problemy z niektórymi przeglądarkami, które wyświetlają przez to krzaczki. Przy wielobajtowym kodowaniu znaków o stałej długości (tzn. takim, że każdy znak jest zapisany za pomocą takiej samej liczby bajtów, np. UTF-16 - 2 bajty, UTF-32 - 4 bajty) pojawia się problem interpretacji kolejności bajtów w znaku. Rozwiązaniem tego problemu jest właśnie BOM.
Znający zagadnienie programiści szczegółowo testowali korzystanie z BOM i UTF-8 w Python 2.3 i stwierdzili:
" Trzeba być ostrożnym przy korzystaniu z BOM i UTF-8. Szczerze mówiąc, myślę, że jest to błąd w Pythonie, ale co zrobić. Python dekoduje wartość BOM w znakach Unicode, zamiast go ignorować !!".

Uwaga:
Jak podano powyżej należy być ostrożnym z stosowaniem do kodowania **BOM** w Python-fu GIMP.
Jeśli edytor który stosujemy, *do tłumaczenia*, daje takę możliwość, to lepiej stosować **Konwertuj na format UTF-8 (bez BOM).** Taką możliwość ma np. **Notepad++.**
Możliwości **Notepad++** są zbliżone do możliwości **SciTE.**
Podstawowa rada jest taka: tworząc wtyczkę, pierwsze ustawić kodowanie, a potem pisać coś w pliku.
Warto też w opcjach przełączyć na stałe UTF-8, przy tworzeniu każdego nowego dokumentu, a nie robić tego ręcznie za każdym razem.
Dlaczego polecam Notepad ++. Jest to darmowy edytor tekstu, który rozpoznaje składnię dla wielu języków programowania (wg. moich obliczeń 50). Można zrobić podświetlanie składni dla wybranego języka, znaleźć parowane nawiasów, podświetla wcięcia, nawiasy i klamry, istnieje automatyczne uzupełnianie, ma zakładki umożliwiające prace na wielu plikach jednocześnie, można zmienić kodowanie dokumentu w ANSI, UCS-2, UTF-8, UTF-8 BOM i wiele więcej m.in. pluginy rozszerzające możliwości, które dostępne są na stronie producenta - i działa w systemach Windows i Linux pod Wine. Pomoże Nam on w znalezieniu błędów w programie oraz tłumaczeniu. http://portableapps.com/apps/development/notepadpp_portable

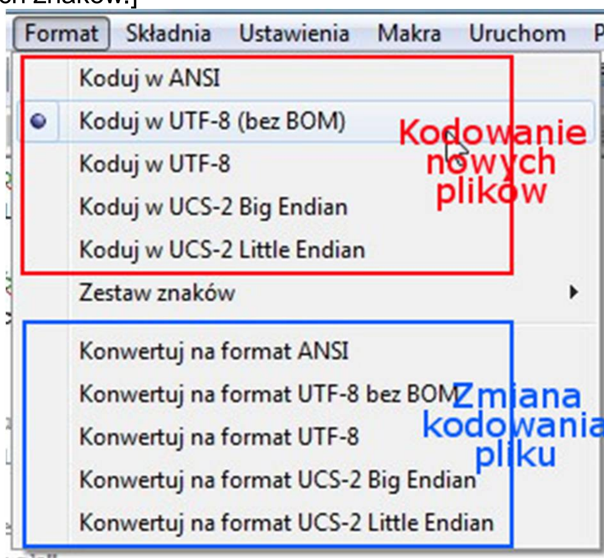
Jeżeli chcemy zapisać plik wtyczki, który jest kodowany w **ANSI** i zawiera polskie znaki to np. stosujemy:
CTRL+A => CTRL+C => "Zmień na UTF-8 bez BOM" => CTRL+V => CTRL+S.

Jeżeli wybierzemy **"UTF-8 bez BOM"** to w menu zmieni się na ANSI, ale zapisuje jak trzeba, w **UTF-8 bez BOM.**

Przykład tworzenia spolszczenia pozyskanego plugina (wtyczki):

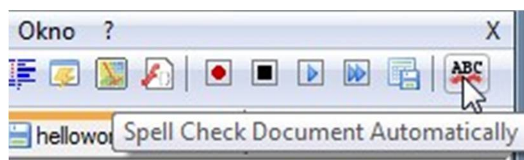
1. Musimy plik poddać edycji np. w edytorze, np. Notepad++ **6.7.8.2** Portable (23 maj 2015) pod Windowsem.
2. Po otwarciu pliku w edytorze, upewniamy się, że wpisywane polskie znaki tłumaczenia napisów są poprawinie wyświetlane. Po zakończeniu tłumaczenia napisów należy zaznaczyć całą zawartość pliku (CTRL+A) i wyciąć go do schowka (CTRL+X).
3. Następnie rozwijamy menu **Format** i klikamy kolejno:
   - "Koduj w ANSI"
   - "Koduj w UTF-8 (bez BOM)"
4. Teraz możemy wkleić zawartość pliku ze schowka (CTRL+V). Polskie znaki nadal powinny być czytelne.
5. Plik po zapisaniu (menu Plik => Zapisz) będzie miał kodowanie UTF-8 bez BOM, dzięki czemu zapisane w nim polskie znaki diakrytyczne będą poprawnie wyświetlane.

[W Notepad++ możemy ustawić w preferencjach programu by każdy nowy dokument był tworzony w formacie **Koduj UTF-8 bez BOM**. Co więcej możemy ustawić by kodowanie to automatycznie zostało ustawione, dla nowo otwieranych plików z formatowaniem **ANSI**, które powoduje problemy (tzw. krzaczki) podczas wyświetlania polskich znaków.]




```
43  register(
44      "python_fu_hello_world",
45      "Obraz Hello world ",
46      "Tworzymy nowy obraz z Naszego łańcucha tekstu",
47      "Akkana Peck",
48      "Akkana Peck",
49      "2010",
50      "Hello world (Py)...",
51      "",        # Tworzenie nowego obrazu, nie działa na istniejacym
52      [
53          (PF_STRING, "string", "Napis", 'Hello, world!'),
54          (PF_FONT, "font", "Wybór czcionki", "Sans"),
55          (PF_SPINNER, "size", "Rozmiar czcionki", 50, (1, 3000, 1)),
56          (PF_COLOR, "color", "Kolor tekstu", (1.0, 0.0, 0.0))
57      ],
58      [],
59      hello_world, menu="<Image>/File/Create")
60
61  main()
```

Podkreślenie pisowni



Klikamy: "Sprawdź automatycznie pisownię dokumentów"
Słownik polski uruchamiamy z menu klikając na ikonkę ABC umieszczoną na Pasku narzędziowym)..

```
43  register(
44      "python_fu_hello_world",
45      "Obraz Hello world ",
46      "Tworzymy nowy obraz z Naszego łańcucha tekstu",
47      "Akkana Peck",
48      "Akkana Peck",
49      "2010",
50      "Hello world (Py)...",
51      "",     # Tworzenie nowego obrazu, nie działa na istniejącym
52      [
53          (PF_STRING, "string", "Napis", 'Hello, world!'),
54          (PF_FONT, "font", "Wybór czcionki", "Sans"),
55          (PF_SPINNER, "size", "Rozmiar czcionki", 50, (1, 3000, 1)),
56          (PF_COLOR, "color", "Kolor tekstu", (1.0, 0.0, 0.0))
57      ],
58      [],
59      hello_world, menu="<Image>/File/Create")
60
61  main()
```

Ten sam fragment wtyczki, po kliknięciu: "Sprawdź automatycznie pisownię dokumentów".
Pierwsze co rzuca się w oczy, to brak podkreślania błędnych wyrazów w samym programie.

DSpellCheck dodatek do sprawdzania pisowni. Od wersji Notepad++ 6.3.3 jest domyślnie dodawany do paczki z edytorem. Rozszerzenie od razu podkreśla błędy w kodzie roboczym, co jest niewątpliwe sporym ułatwieniem.

Dalej:
http://www.crimsteam.site90.net/crimsteam/artykuly/artykuly_webtips_npp_instalacjaorazkonfiguracja.html
http://www.crimsteam.site90.net/crimsteam/artykuly/artykuly_webtips_npp_sprawdzaniepisowni.html#spell-checker,1,13
https://www.youtube.com/watch?v=ZSTl0RXc4nk Style konfigurator
https://www.youtube.com/watch?v=PuTOOrRYTZA !!!!!!
https://www.youtube.com/watch?v=rw_VrS-McvM
https://www.youtube.com/watch?v=PzjPu5F9K9Y Notepad++ Tip and Trck

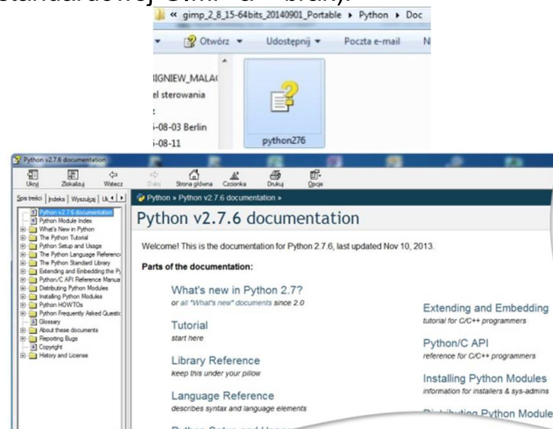
Można również korzystać z PyScripter
https://www.youtube.com/watch?v=qwWiIW3z-Vo Jak korzystać PyScripter przenośny IDE Pythona**…**
http://www.windows8downloads.com/win8-pyscripter-portable-guedvsmg/ 13 Maj, 2015
http://portablepython.com/wiki/PortablePython2.7.6.1/ lub 3.2.5.1

To co powiedziano powyżej pomija zagadnienie:

Jaki jest właściwy sposób na umiędzynarodowienie wtyczki Pythona.
Niektóre wtyczki obsługują wiele języków, korzystając z zewnętrznego (zewnętrznych) pliku(ów) zawierających bazę wszystkich tekstów występujących w danej wtyczce.
Są to pliki paczki językowej – **pot** plik utworzony przez autora umożliwiający stworzenie plików **po** i **mo**
Istnieją specjalne programy np. **PoEdit** przeznaczone dla tłumaczy, pozwalające na tworzenie w/w plików.
Poedit jest to darmowy program, który pozwala w prosty sposób tłumaczyć pliki **.po** i generować pliki **.mo**.
Poedit jest dostępny dla wszystkich popularnych systemów operacyjnych, można pobrać klikając tutaj
Przykład zastosowania podano w Poradniku:
http://zbyma.gimpuj.info/Poradnik%20-%20T%C5%82umaczenia%20wtyczek%20GIMP%2C%20czyli%20jak%20wygenerowa%C4%87%20plik%20pot%20%20korzystaj%C4%85c%20z%20PoEdit.pdf
lub:
http://1drv.ms/1j3KbWD
Tyle *pobieżnego* wprowadzenia do języka Python, niezbędnego do odczytywania wtyczek, więcej w literaturze – wykaz poszukiwanej załączono np.:

Gdzie szukamy więcej informacji

Kompilacja Partha *GIMP 2.8.15 64bits Portable* zawiera w sobie dokumentację Python 2.7.6 z której możemy korzystać (w wersji standardowej GIMP-a – brak):



Dalej:

http://python.edu.pl/byteofpython/2.x/04.html **Pierwsze kroki**

https://wiki.python.org/moin/PythonBooks

Opis poszczególnych elementów programowania możemy odnaleźć w dokumentacji dostępnej pod adresem https://docs.python.org/2/tutorial/ http://www.python.org (info dostępna online, do pobrania w pdf). Liczba publikacji opisujących język Python w naszym kraju nie jest zachwycająca. O ile z dostępnością ebooków oraz kursów nie jest źle (np. http://pl.python.org/kursy,jezyka.html) to problemem jest to, że nie znajdziemy w nich charakterystyki elementów Pythona oraz to, że zawierają one opis starszych wersji Pythona.

http://python.edu.pl/index.html Wszystkie materiałye edukacyjne dotyczące Pythona w jednym miejscu (z myślą o nowicjuszach!.)

Podręcznik: Python.- wprowadzenie.- wydanie-IV pełna wersja.pdf – Helion (można ściągnąć)

http://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie Podręcznik można pobrać jako pdf.

https://pl.python.org/view/kursy/lwdp.pdf Lamerskie Wprowadzenie do Pythona.

http://szgrabowski.kis.p.lodz.pl/Python-podstawy.html Poradnik dla początkujących

https://pl.python.org/view/kursy/przewodnik.pdf Przewodnik po języku Python Wyd. 2.3

http://brain.fuw.edu.pl/edu/TI:Programowanie_z_Pythonem/Wersja_do_druku

http://shallowsky.com/python/ 9 lekcji Pythona **!!!**

https://docs.python.org/2/tutorial/index.html !!!!!!

http://www.python.rk.edu.pl/w/p/wprowadzenie-do-pythona/

http://python101.readthedocs.org/pl/rtd/basic1/basic1.html !!!!!!

http://pl.wikipedia.org/wiki/Python

http://www.jamesh.id.au/software/pygimp/pygimp.html

http://docs.gimp.org/en/gimp-scripting.html#gimp-concepts-plugins

http://www.gimp.org/docs/python/

http://developer.gimp.org/api/2.0/libgimp/index.html

http://developer.gimp.org/api/2.0/libgimp/libgimp-gimpimage.html#gimp-image-scale

http://docs.gimp.org/en/gimp-filters-python-fu.html

https://git.gnome.org/browse/gimp/tree/plug-ins/pygimp/gimpfu.py

http://www.ucs.cam.ac.uk/docs/course-notes/unix-courses/PythonProgIntro

https://www.youtube.com/watch?v=M0IrGCyQAJA [Tutorial] Writing a Basic Gimp Plugin in Python

https://www.youtube.com/watch?v=DGDUGDxnGpI [Tutorial] Introduction to PythonFu Objects, and Iterating Layers,

https://www.youtube.com/watch?v=Z-_B2lqrvpY devICT - Automating Gimp Using Python **!!!**

oraz szereg innych…

http://www.jamesh.id.au/software/pygimp/pygimp.html#INTRODUCTION **!!**

http://www.gimp.org/tutorials/Automate_Editing_in_GIMP/

Uwaga:

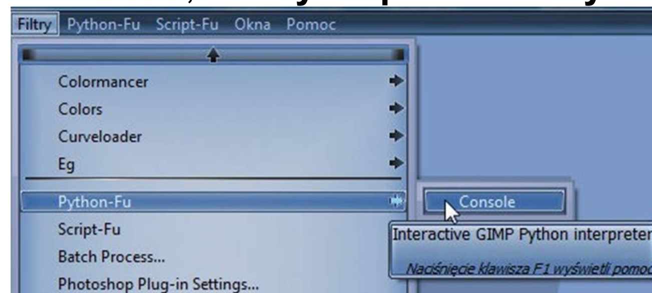
Twórcy Pythona nie przejmują sie raczej tzw. zgodnością wsteczną - programy (polecenia, instrukcje) dopasowane do starszej wersji Pythona mogą nie działać w nowszej.

W internecie jest sporo informacji o Pythonie. Z powodu częstych aktualizacji języka i braku zgodności wstecznej łatwo trafić na kursy i tutoriale niezgodne z wersją 2.7, a tym bardziej z najnowszą wersją 3.2. Trzeba się liczyć z koniecznością zmian w przytaczanych kodach źródłowych. Najpewniejszym źródłem jest oficjalna strona http://www.python.org/doc/ zawierająca dokumentację i tutoriale.

[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.]

Brak zgodności w jeszcze większym stopniu dotyczy książek.
Najlepszą chyba książką jest Zanurkuj w Pythonie (Dive into Python). (oczywiście mogę się mylić)

Konsola Pythona w GIMP-ie, "Filtry => podmenu Python-Fu"



Rysunek "podmenu Python-Fu "

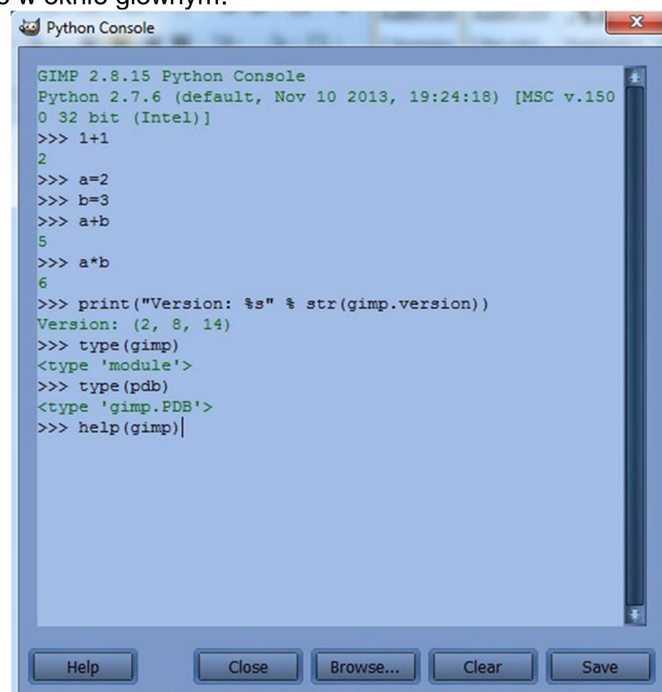
Domyślnie to podmenu zawiera tylko **Console** Python-Fu.
Python-Fu jest zestawem modułów *Python Programming Language*. http://www.python.org , które działają jako powłoka do `libgimp` umożliwiając pisanie wtyczek dla GIMP-a.

Aktywacja podmenu

- Jak widać uzyskujemy dostęp do tego polecenia z menu obrazu **Filtry => Python-Fu => Console**

Konsola Python-Fu

Zapewnia ona możliwość szybkiego testowania procedur i wyświetlania ich wyników przed rozpoczęciem pisania właściwego dodatku. Konsola jest znakomitym sposobem sprawdzania języka Python i przyzwyczajania się do jego składni.
Konsola Python-Fu jest uruchomiana jako okno dialogowe " powłoki Python" (interpreter Pythona w trybie interaktywnym). Konsola jest skonfigurowana do korzystania z procedur wewnętrznej biblioteki `libgimp` GIMP-a.
Możemy użyć konsoli Python-Fu aby interaktywnie testować polecenia Pythona.
Konsola składa się z dużego przewijanego okna głównego dla wejścia i wyjścia, w którym można wpisać polecenia Pythona.
[Moduł **gimpfu** podłącza się automatycznie przy uruchomieniu konsoli, dlatego nie należy niczym się niepokoić i wprowadzać polecenia.]
Po wpisaniu polecenia Pythona, a następnie naciśnięciu klawisza **Enter**, polecenie jest wykonywane przez interpreter Pythona. Wyjście polecenia, jak również jego wartość zwrotna (i komunikat o błędzie, jeśli taki istnieje) będą wyświetlane w oknie głównym.



Rysunek: Konsola Python-Fu

Wejście rozróżnione zostało poprzez obecność znaku zachęty (">>>" **ang**. ***Prompt***, oznacza, że Python oczekuje na wprowadzenie polecenia).
W dalszych przykładach wejście i wyjście rozróżnione zostało poprzez obecność znaków zachęty ("**>>>** " i "**...** "): aby powtórzyć przykład, trzeba przepisać wszystko za znakiem zachęty; linie, które się nim nie zaczynają, **są wyjściem** (odpowiedziami) interpretera.

Uwaga! Pojawienie się po raz drugi znaku zachęty oznacza, że trzeba wprowadzić pusty wiersz. Wtedy następuje koniec instrukcji wielowierszowej.

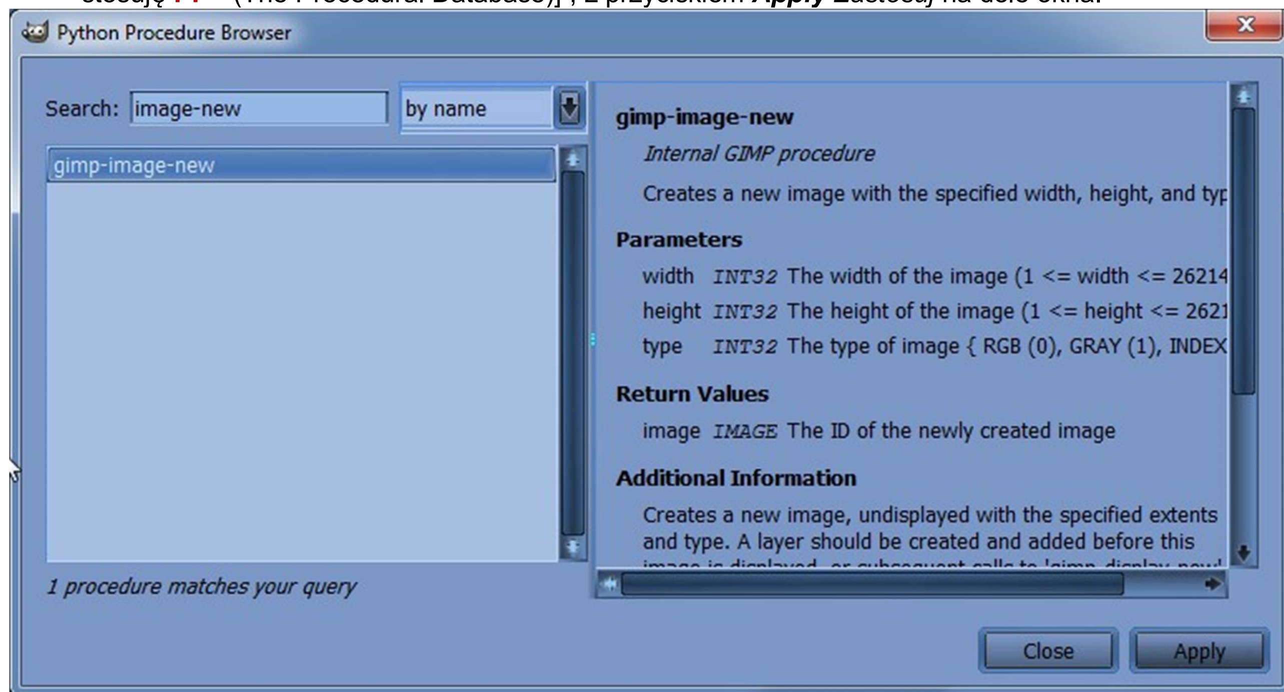
Wiele przykładów w tym poradniku, nawet te wprowadzone w linii poleceń, zawierają komentarze.
W Pythonie komentarz rozpoczyna się od znaku "**#**" i ciągnie się aż do końca fizycznego wiersza.
Komentarz może pojawić się na początku linii lub kończyć instrukcję. (dalej informacja rozszerzona)
W konsoli Python-fu można pracować w trybie **interaktywnym** lub **skryptowym**. Ten pierwszy tryb jest bardzo poręczny do nauki języka – wpisujemy polecenie (*z palca*) i od razu widzimy efekt.
Wpisujemy: 1 + 1 (klikamy Enter). Wyświetlony wynik **2**. Działa!

Ten sam efekt można otrzymać pisząc instrukcję: `print 1+1`. Ale także `print(1+1)`.[biały znak, spacja - zastąpiony nawiasami]
Jednak w trybie interaktywnym na ekranie jest wyświetlana wartość wyrażenia i słowo `print` nie jest koniecznie.

Polecenia `dir(...)` i `help(...)`, pozwalają uzyskać najwięcej informacji.
Polecenie **dir**. wyświetli nam zawartość katalogu,(*nie jest poleceniem Unix*)

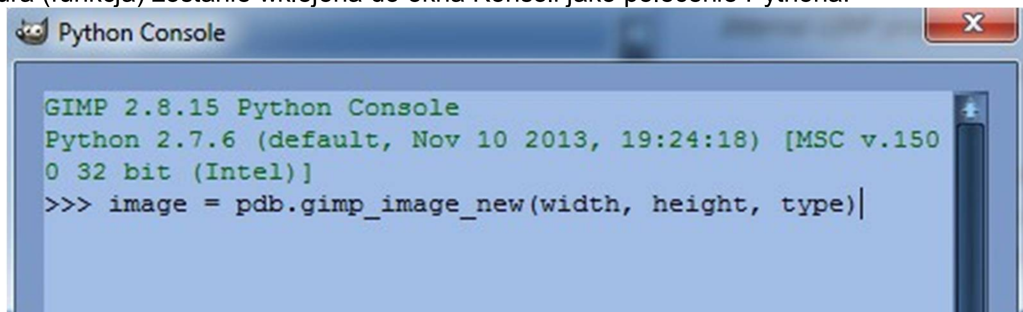
Przyciski konsoli Python-Fu

Save *Zapisz*
To polecenie pozwala zapisać zawartość głównego okna, to jest wejście i wyjście konsoli Python-Fu (włączając w tym symbole "**>>>**").
Clear *Wyczyść*
Po kliknięciu na ten przycisk, zawartość głównego okna zostanie usunięta. Pamiętaj, że nie możesz przywrócić usuniętej zawartości przy użyciu polecenia Save - Zapisz.
Browse *Przeglądaj*
Po kliknięciu przycisku otworzy się okno Python Procedure Browser - [Przeglądarka_Procedur, dalej stosuję **PP** - (The Procedural Database)] , z przyciskiem ***Apply*** *Zastosuj* na dole okna.



W oknie **Procedure Browser PP** wyszukujemy interesującą Nas procedurę wpisując w **Search** S*zukaj*: np. **image-new** (stosujemy łączniki) lub tylko **new**
[Możemy wyszukać interesujący nas skrypt za pomocą nazwy, opisu skryptu, zawartej w nim pomocy, autora, praw autorskich, daty otworzenia/edycji oraz typu.
Po wyborze jakiejkolwiek wbudowanej procedury po prawej stronie okna pokazuje się jej opis.

Do czego służy procedura, ile parametrów potrzebuje, co zwraca oraz krótka notka informująca o autorze, dacie utworzenia oraz prawach autorskich.
Podczas pisania własnych dodatków okno *Przeglądarki procedur* stanowi bezcenne narzędzie, ponieważ pozwala dokładnie stwierdzić, jakie są dostępne funkcje programu GIMP.]
Po naciśnięciu w Procedure Browser przycisku ***Apply***, lub dwuklikiem na nazwie, wywołana wybrana procedura (funkcja) zostanie wklejona do okna Konsoli jako polecenie Pythona:



Zastosowana procedura z **PP**

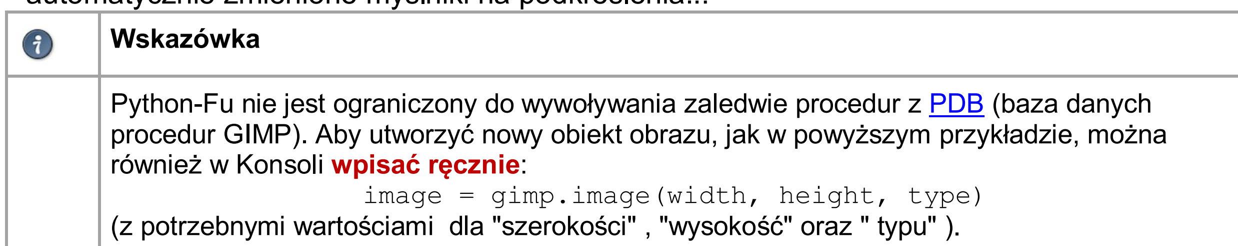
Teraz po prostu musimy wymienić nazwy parametrów na ich wartości
(powyżej: "width - szerokość" , "height - wysokość" i "type") muszą mieć określoną wartość (domyślnie w pixelach):

```
image = pdb.gimp_image_new(400, 300, RGB)
```

Następnie naciskamy na klawiaturze klawisz **Enter**, aby wykonać procedurę.

Możemy (i należy!) korzystać z stałych, *które można znaleźć w opisie parametrów procedury*, na przykład "RGB-IMAGE" lub "OVERLAY-MODE".

Należy pamiętać:
gdy wklejamy do okna Konsoli procedurę (funkcję) z **PP,** klikając w niej **Apply** zostaną automatycznie zmienione myślniki na podkreślenia.!!



	Wskazówka
	Python-Fu nie jest ograniczony do wywoływania zaledwie procedur z PDB (baza danych procedur GIMP). Aby utworzyć nowy obiekt obrazu, jak w powyższym przykładzie, można również w Konsoli **wpisać ręcznie**: `image = gimp.image(width, height, type)` (z potrzebnymi wartościami dla "szerokości" , "wysokość" oraz " typu").

Należy jednak pamiętać, że *przy ręcznym wpisywaniu procedury* (funkcji), trzeba wymienić myślniki - *łączniki* (" **-** ") na podkreślenia (" _ "):
RGB_IMAGE, OVERLAY_MODE, dlatego, że dla Pythona "–" oznaczałoby odejmowanie, i nie jest dozwolone w nazwach stałych funkcji.

Procedury, czy funkcje *wbudowane* (biblioteczne) to funkcje, które są zawsze dostępne do wywołania.

Close
Naciśnięcie tego przycisku zamyka Konsolę.

Wniosek jest taki, że nawet osoba która nie jest programistą. Może tworzyć proste wtyczki, posługując się oknem Przeglądarka procedur **PP** programu GIMP w celu wyszukania funkcji nadających się do zastosowania w wtyczce i eksperymentowania z językiem Python za pośrednictwem interaktywnej Konsoli.

Module gimp

Listing 1 Moduły gimp

```
>>> help(gimp)
```

Kiedy klikniemy Enter zostaną wyświetlone wszystkie funkcje i klasy dostępne dla gimpa w module Pythona. Dodatkowo dostaniemy informacje odnośnie miejsca zapisu naszego programu oraz numer aktualnej wersji.
Uwaga: *klawiszami strzałek na klawiaturze, można kopiować poszczególne linie kodu*

Python Console

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
>>> help(gimp)
Help on module gimp:

NAME
    gimp - This module provides interfaces to allow you to write gimp plugins

FILE
    c:\users\zbigniewmalach\desktop\gimpportab\gimp_2_8_15-64bits_20140901_po
rtable\lib\gimp\2.0\plug-ins\gimp.pyd

CLASSES
    __builtin__.object
        Display
        Image
        Item
            Drawable
                Channel
                Layer
                    GroupLayer
            Vectors
        Parasite
        PixelFetcher
        PixelRgn
        Tile
    exceptions.RuntimeError(exceptions.StandardError)
```

Help Close Browse... Clear Save

```
>>> dir(gimp)
```

Python Console

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
>>> dir(gimp)
['Channel', 'Display', 'Drawable', 'GroupLayer', 'Image', 'Item', 'Layer
', 'Parasite', 'PixelFetcher', 'PixelRgn', 'Tile', 'Vectors', 'VectorsBez
ierStroke', '_PyGimp_API', '__doc__', '__file__', '__name__', '__package_
_', '_id2display', '_id2drawable', '_id2image', '_id2vectors', 'attach_ne
w_parasite', 'check_size', 'check_type', 'checks_get_shades', 'context_ge
t_gradient', 'context_pop', 'context_push', 'context_set_gradient', 'data
_directory', 'default_display', 'delete', 'directory', 'display_name', 'd
isplays_flush', 'displays_reconnect', 'domain_register', 'error', 'exit
', 'extension_ack', 'extension_enable', 'extension_process', 'fonts_get_l
ist', 'fonts_refresh', 'gamma', 'get_background', 'get_data', 'get_foregr
ound', 'get_progname', 'gradient_get_custom_samples', 'gradient_get_unifo
rm_samples', 'gradients_get_gradient', 'gradients_get_list', 'gradients_s
ample_custom', 'gradients_sample_uniform', 'gradients_set_gradient', 'gtk
rc', 'image_list', 'install_procedure', 'install_temp_proc', 'locale_dire
ctory', 'main', 'menu_register', 'message', 'monitor_number', 'parasite_a
ttach', 'parasite_detach', 'parasite_find', 'parasite_list', 'pdb', 'pers
onal_rc_file', 'plug_in_directory', 'progress_init', 'progress_install',
'progress_uninstall', 'progress_update', 'quit', 'register_load_handler
', 'register_magic_load_handler', 'register_save_handler', 'set_backgroun
d', 'set_data', 'set_foreground', 'show_help_button', 'show_tool_tips', '
sysconf_directory', 'tile_cache_ntiles', 'tile_cache_size', 'tile_height
', 'tile_width', 'uninstall_temp_proc', 'user_directory', 'vectors_import
_from_file', 'vectors_import_from_string', 'version', 'wm_class']
>>>
```

Help Close Browse... Clear Save

Można uzyskać wbudowaną dokumentację na temat API przy użyciu `dir(GIMP)` w powłoce Python Gimp. Pierwszą należy zaimportować gimpfu.

Innym sposobem jest użycie `help()` w określonej zmiennej, na przykład layer.

Gimp Procedurial DataBase => Module gimp.pdb

Listing 2 – Moduły gimp.pdb

```
>>> dir(gimp.pdb)
```

Wtedy widzimy listę:

```
Python Console

GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> dir(gimp.pdb)
['1_motif_pattern_samj', 'Bokeh_Effect_version2', 'Butte
rfly_3_Lylejk_samj', 'Butterfly_Wings_by_Lylejk', 'Cadre
_Volute_et_Cadre', 'CamoTruck_Net_Blob', 'CamoTruck_Net_
Colors', 'CamoTruck_Net_Grass', 'CamoTruck_Net_Sprawl',
'Colorful_Light_Effect', 'Contour_Pie_Pans_samj', 'Cossi
n_En_Spirale_samj', 'Cossins_Circulaires', 'Denim', 'Epi
cycloid_for_beginners_samj', 'FU_BSSS', 'FU_ColorSaturat
ion', 'FU_Copyright', 'FU_HighPassSharpen', 'FU_fog', 'F
U_frame_hover', 'FU_frame_negative', 'FU_frame_poster',
'FU_gimp_reflection', 'FU_glass_selection', 'FU_grey_poi
nt', 'FU_highpass_image', 'FU_lomo', 'FU_midtone_sharp
', 'FU_old_paper', 'FU_pastel_image', 'FU_pastel_portrai
t', 'FU_pencil_sketch_BW', 'FU_picture_to_graphic', 'FU_
quick_sketch', 'FU_shadows_highlights', 'FU_sharp_blur
', 'FU_smart_sharpening', 'FU_soft_focus', 'FU_stair_res
ize', 'FU_water_reflection', 'FU_watermark', 'FU_web_pho
to_editor', 'Fix_CA', 'Glossy_Metal_3D_Text_By_Monsoonam
i', 'Grunge_Stamp', 'Hypocycloid_for_beginners_samj', 'L
es_chemins_colores_samj', 'Motif_Geometrique_1_3D_samj
', 'Motif_Spirale_samj', 'Motif_Volute_et_Cadre', 'My_Fl
owers_samj', 'Pattern_From_Image_212_512_samj', 'Pursuit
_Curve_samj', 'RMA_path_blend', 'Round_Web_2_0_Button_wi
th_a_Metal_Ring', 'Script_Fu_Masques_samj', 'Selection_P

Help   Close   Browse...   Clear   Save
```

Wyszczególnienie gimpenum

Listing 3 – pomoc dla GimpEnum

```
>>> import gimpenums
>>> help(gimpenums)
```

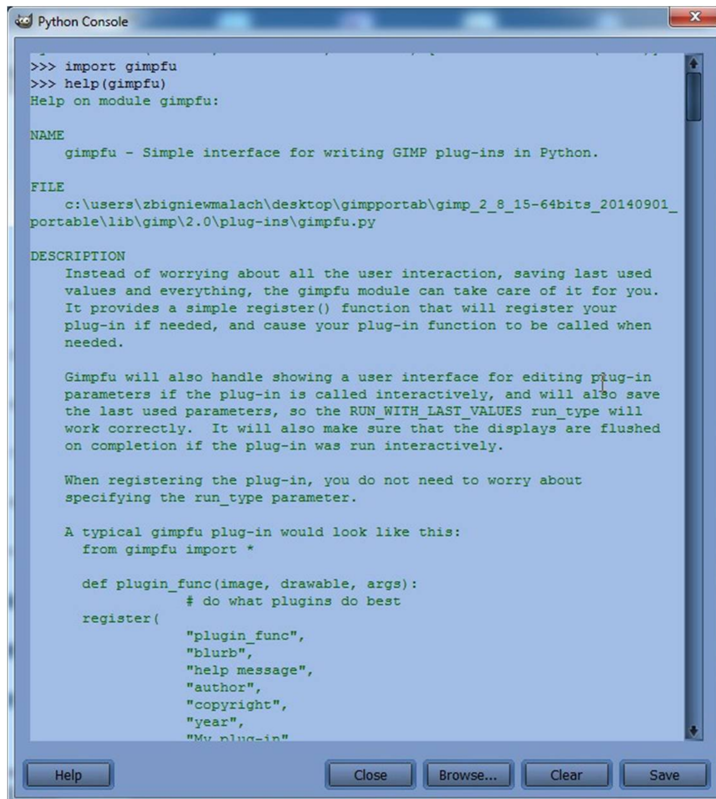
Zobaczymy szczegółowe wartości przydatnych stałych

```
Python Console

>>> help(gimpenums)
Help on module gimpenums:

NAME
    gimpenums

FILE
    c:\users\zbigniewmalach\desktop\gimpportab\gimp_2_8_15-64bits_20140901
_portable\lib\gimp\2.0\plug-ins\gimpenums.py

DESCRIPTION
    #   Gimp-Python - allows the writing of Gimp plugins in Python.
    #   Copyright (C) 2005  Manish Singh  <yosh@gimp.org>
    #
    # 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 3 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, see <http://www.gnu.org/licenses/
>.

DATA
    ABSOLUTE_CONVOL = 1
    ADDITION_MODE = 7
    ADD_ALPHA_MASK = 2
    ADD_ALPHA_TRANSFER_MASK = 3
    ADD_BLACK_MASK = 1
    ADD_CHANNEL_MASK = 6
    ADD_COPY_MASK = 5
    ADD_SELECTION_MASK = 4
    ADD_WHITE_MASK = 0
    ALL_HUES = 0

Help   Close   Browse...   Clear   Save
```

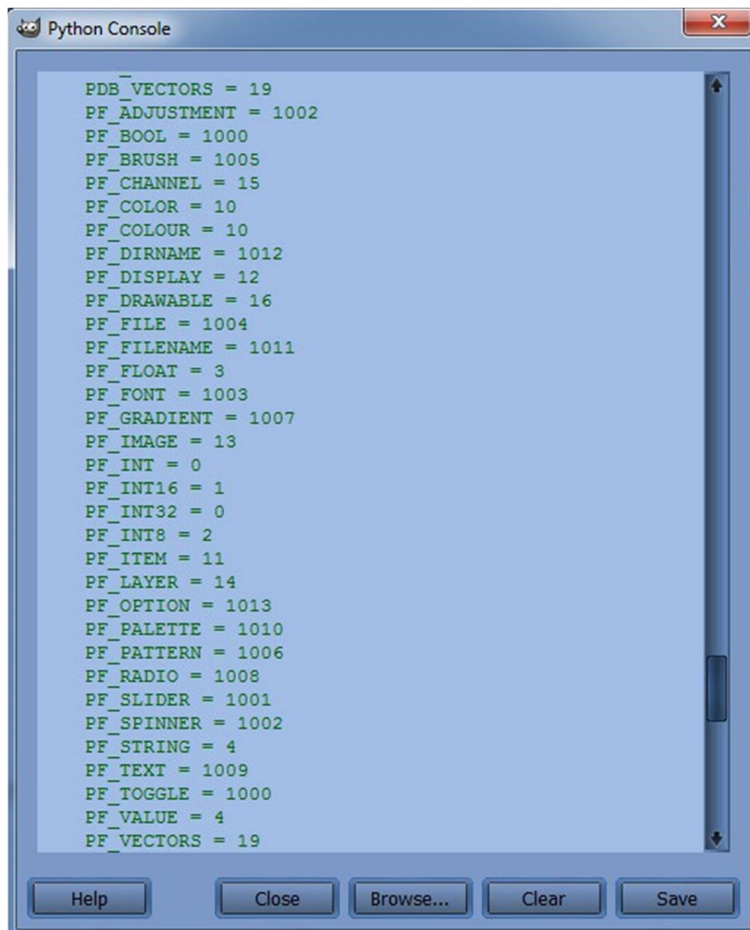
Module gimpfu

Listing 4 Moduły gimpfu

```
>>> import gimpfu
>>> help(gimpfu)
```



Cd:



```
>>> import gimpfu
>>> help(str)
Help on class str in module __builtin__:

class str(basestring)
 |  str(object='') -> string
 |
 |  Return a nice string representation of the object.
 |  If the argument is a string, the return value is th
e same object.
 |
```

Kiedy chcemy przeglądnąć zawartość modułów w Pythonie z pomocą przychodzą nam dwie ważne funkcje - **`dir`** oraz **`help`**.
Za pomocą funkcji '**`dir`**' możemy zobaczyć, jakie funkcje zostały umieszczone w dowolnym module.
Aby poznać listę dostępnych wartości parametrów PF, wprowadzamy w konsoli Pythona Gimp

```
>>> import gimpfu
>>> dir(gimpfu)
```

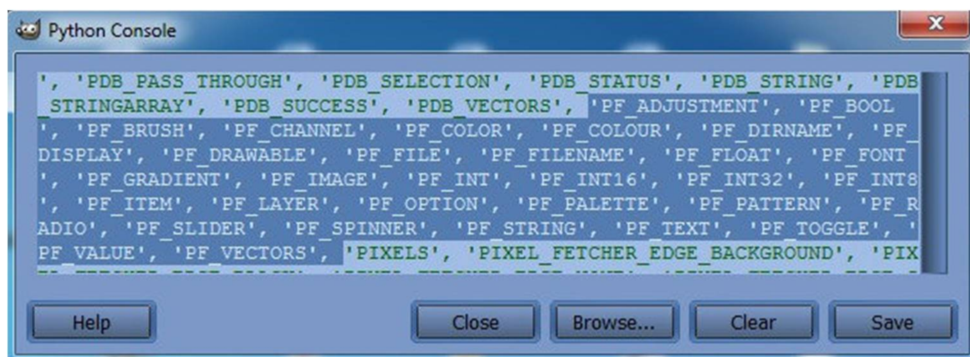
Wtedy widzimy pojawiającą się następującą listę:

Python Console

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> import gimpfu
>>> dir(gimpfu)
['ABSOLUTE_CONVOL', 'ADDITION_MODE', 'ADD_ALPHA_MASK', '
ADD_ALPHA_TRANSFER_MASK', 'ADD_BLACK_MASK', 'ADD_CHANNEL
_MASK', 'ADD_COPY_MASK', 'ADD_SELECTION_MASK', 'ADD_WHIT
E_MASK', 'ALL_HUES', 'ALPHA_CHANNEL', 'BACKGROUND_FILL
', 'BEHIND_MODE', 'BG_BUCKET_FILL', 'BLUE_CHANNEL', 'BLU
E_HUES', 'BLUR_CONVOLVE', 'BRUSH_GENERATED_CIRCLE', 'BRU
SH_GENERATED_DIAMOND', 'BRUSH_GENERATED_SQUARE', 'BRUSH_
HARD', 'BRUSH_SOFT', 'BURN', 'BURN_MODE', 'CHANNEL_OP_AD
D', 'CHANNEL_OP_INTERSECT', 'CHANNEL_OP_REPLACE', 'CHANN
EL_OP_SUBTRACT', 'CHECK_SIZE_LARGE_CHECKS', 'CHECK_SIZE_
MEDIUM_CHECKS', 'CHECK_SIZE_SMALL_CHECKS', 'CHECK_TYPE_B
LACK_ONLY', 'CHECK_TYPE_DARK_CHECKS', 'CHECK_TYPE_GRAY_C
HECKS', 'CHECK_TYPE_GRAY_ONLY', 'CHECK_TYPE_LIGHT_CHECKS
', 'CHECK_TYPE_WHITE_ONLY', 'CLIP_TO_BOTTOM_LAYER', 'CLI
P_TO_IMAGE', 'COLOR_ERASE_MODE', 'COLOR_MODE', 'CONSOLE
', 'CUSTOM_MODE', 'CUSTOM_PALETTE', 'CYAN_HUES', 'Cancel
Error', 'DARKEN_ONLY_MODE', 'DESATURATE_AVERAGE', 'DESAT
URATE_LIGHTNESS', 'DESATURATE_LUMINOSITY', 'DIFFERENCE_M
ODE', 'DISSOLVE_MODE', 'DIVIDE_MODE', 'DODGE', 'DODGE_MO
DE', 'ERROR_CONSOLE', 'EXPAND_AS_NECESSARY', 'EXTENSION
', 'FALSE', 'FG_BG_HSV_MODE', 'FG_BG_RGB_MODE', 'FG_BUCK
ET_FILL', 'FG_TRANSPARENT_MODE', 'FIXED_DITHER', 'FLATTE
```

Help | Close | Browse... | Clear | Save

Kiedy już znajdziemy funkcję, której chcemy użyć, możemy dowiedzieć się o niej więcej używając '**`help`**' w interpreterze Pythona.



Aby zlokalizować katalog z GIMP-em:
Wprowadzamy do Pyton-Fu => Console:
`print gimp.directory`
I pojawi się ścieżka, gdzie jest nasz katalog:

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> print gimp.directory
C:\Users\ZbigniewMalach\Desktop\GIMPPortab\gimp_2_8_15-6
4bits_20140901_Portable\Preferences
>>> |
```

Przykład mojego.

Importowane Podstawowe moduły GIMP-a

Moduły to, biblioteki funkcji, których możemy używać w swoich programach.
Moduły Pythona to **obiekty**, które posiadają kilka przydatnych atrybutów.
Aby nasza wtyczka Pythona rozpoczęła zarządzać GIMP-em, pociągać za nitki jego **API** (*Application Programming Interface*), niezbędne jest podłączenie do niej odpowiednich modułów (powyżej były zrzuty):

- **gimpfu** — to postawowy moduł, który zapewnia prosty interfejs do pisania wtyczek, podobne do tego, co zapewnia skrypt-fu. Zapewnia **GUI** (**G**raphical **U**ser **I**nterface - Graficzny Interfejs Użytkownika) do wprowadzania parametrów *w trybie interaktywnym* i wykonuje kilka testów poprawności podczas rejestracji wtyczki. Zawiera w sobie funkcję **register()** (rejestruje dodatek w PDB).
  Za pomocą modułu **"from gimpfu import *"**, można w łatwy sposób uzyskać wszystkie powszechnie używane symbole należące do przestrzeni nazw wtyczki.
- **main()** [uruchamia wtyczkę (dodatek)], niezbędne stałe i zmienne otoczenia, a także podłącza szereg modułów, niezbędnych do pracy
- **gimp** — tu zebranę są podstawowe procedury, funkcje i struktury danych, włączając obiekty Image, Layer i zmienną **pdb** dla dostępu do ***The Procedural Database***. Moduł automatycznie podłącza się przy podłączeniu modułu **gimpfu**
- **gimpenums** — przydatne stałe, Moduł automatycznie podłącza się przy podłączeniu modułu **gimpfu**
- **gimpui** — ten moduł zawiera w sobie elementy **UI**, zawarte w bibliotece **libgimpui**, pomagają w pracy z elementami graficznego interfejsu
- **gimpshelf** — pozwala dodatkowo zapamiętać dowolne dane w czsie pracy z obrazem i przechowywać je do czasu zamknięcia okna GIMP (takie seansowe ciasteczka tylko dla GIMP-a)
- **gimpplugin** — alternatywny moduł, zapewniający większą elastczność, ale mniejsze możliwości, niż standardowy **gimpfu**. Jeśli wyobrazimy sobie, że **gimpfu** — to zestaw Lego, zbiór wcześniej przygotowanych narzędzi, z których należy budować dodatek, to **gimpplugin** — jest plastyczną gliną, z której można ulepić wszystko, co się chce, ale całą brudną pracę, odpowiednio trzeba będzie wykonać samemu
  Zmienna **pdb** – jest zmienną dostępu do bazy danych proceduralnych. Dla wygody, jest importowana z gimpfu należy do przestrzeni nazw wtyczki.

Wskazówka:
Jeśli nazwa funkcji zaczyna się od słowa "**gimp**" - to są to "natywne - rodzime" funkcje GIMP-a**,**
"**plug-in**" - to zwykle kompilowane wtyczki napisane w języku "C" (najczęściej filtry),
"**script-fu**" – to rozszerzenia pisane w **Scheme**,
a "**python-fu**" – to wtyczki (dodatki) napisane w Pythonie.

Dalsze informacje szukamy w:
http://www.gimp.org/docs/python/index.html m.in. **moduły i funkcje GIMP-Python i jak z nich korzystać** oraz
http://developer.gimp.org/api/2.0/libgimp/index.html
http://developer.gimp.org/api/2.0/libgimp/libgimpui.html
http://developer.gimp.org/api/2.0/libgimp/libgimp-index-new-in-2-8.html
http://developer.gimp.org/api/2.0/app/app-plug-in-part.html itd.

Kod źródłowy modułów można znaleźć także w plikach, znajdujących się w::
/usr/lib/gimp/2.0/python — jeśli pracujemy w środowisku GNU/Linux, lub
C:\Program Files\Gimp-2.8\lib\gimp\2.0\python — jeśli pracyjemy w środowisku Windows.

Ad meritum

Trzeba, absolutnie i kategorycznie, dużo programować (przeklajanie kodu się nie liczy **!!**).
Można stosować naukę poprzez modyfikację istniejących wtyczek, często jesteśmy w stanie osiągnąć ciekawe rzeczy po zaledwie kilku dniach pracy.

Tworzenie obrazu za pomocą konsoli

W celu zapoznania się z metodą spróbujemy najprostszej rzeczy: utworzymy nowy obraz o zadanych wymiarach, wypełniony określonym kolorem.

Tworzyć obraz, musimy się zastanowić, jakie czynności powinny być wykonanie i w jakiej kolejności.
1. Chcemy mieć obraz (i ewent. okno: warstwy, kanały, ścieżki itp.).
Co musimy wiedzieć o naszym obrazie? Ważne będą Wymiary i typ (RGB, skala szarości, indeksowany)
2. Następnie, w poniższym przykładzie, będziemy potrzebować warstwę, więc możemy ją dodać.
Co może być ważne w warstwie? Wymiar, krycie, tryby mieszania, itp. (omówimy to później, ale należy o tym pamiętać!)
3. Utworzyliśmy nasz obraz i warstwę, a co ma być w tej warstwie? Jeszcze nic. Musimy powiedzieć, GIMP, czy ma coś dodać. W naszym przykładzie chcemy wypełnić obraz kolorem. Dlatego musimy wskazać w **Konsoli Python-Fu** jak ma to zrobić.
4. Czy jeśli stworzymy nasz obraz i dodamy do niego warstwę to automatycznie pojawi się na ekranie? Jeśli robimy to w interfejsie GIMP-a, odpowiedź brzmi "tak", ale z **Konsoli Python-Fu**, odpowiedź brzmi "**nie**", obraz będzie tylko w pamięci. Musimy podpowiedzieć GIMP, aby pokazał nam wynik.

Określone powyżej zadania zrobimy w kilka minut. Będzie to kilka kroków, które zrobimy, aby osiągnąć nasz cel, przy czym nie będziemy robić ich w identycznej kolejności, jak opisane powyżej. Przede wszystkim chodzi o to aby pokazać, co, jak i dlaczego tak robimy.

A więc rozpoczynamy.

GIMP-a mamy uruchomionego, okno obrazów jest puste, zaczynamy proces tworzenia obrazu.
Cały czas będziemy korzystać z **Konsoli Python-Fu** oraz **Przeglądarki Procedur (PP).**

W pierwszym kroku rozpoczynającym tworzenie obrazu jest nam potrzebna procedura tworzenia obrazu.

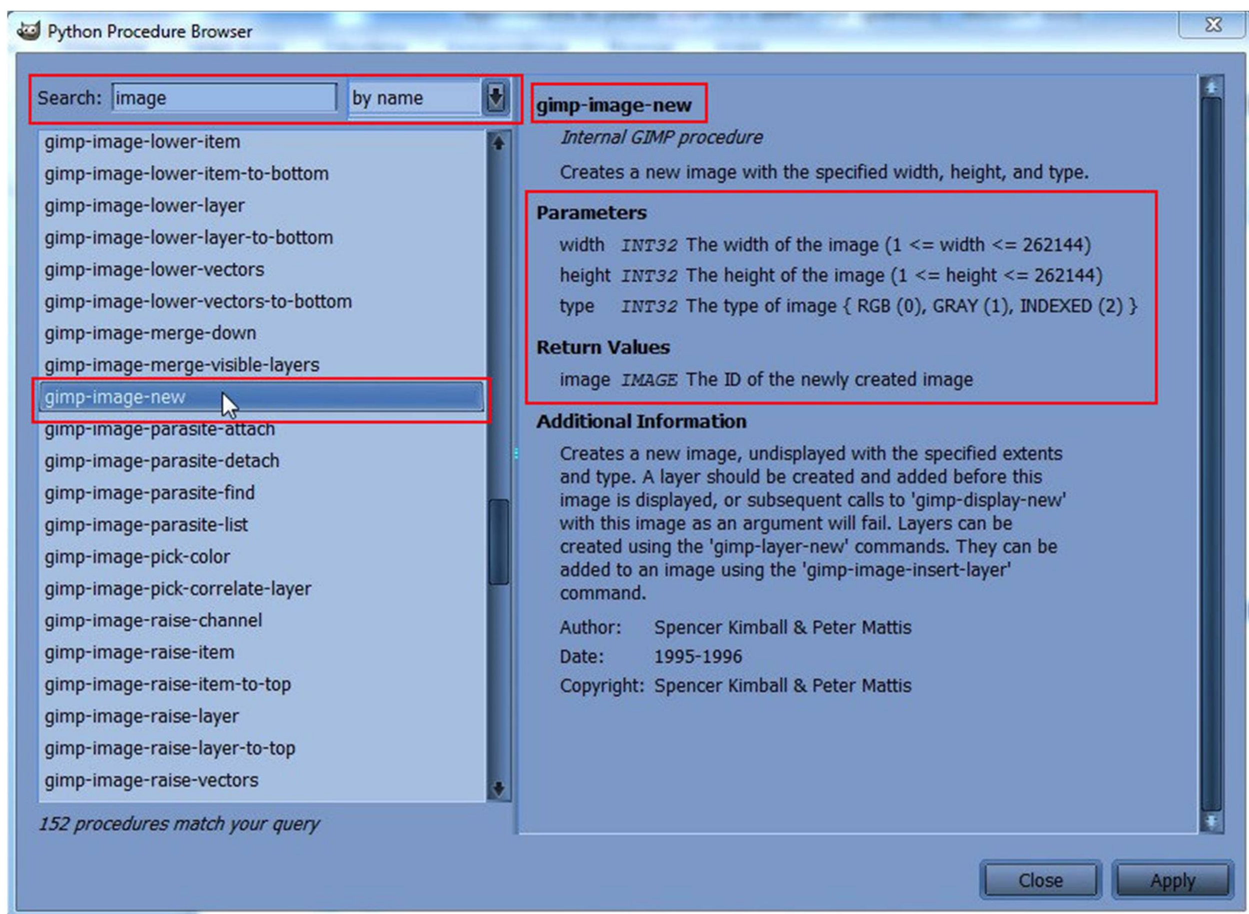
`gimp-image-new`

W tym celu w oknie **Edytor obrazów GIMP**-a wybieramy z menu:

Filtry => Python-Fu => Console.

Po chwili otworzy się okno **Python Console** (dalej będę stosował skrót nazwy **KPF - Konsola Python-Fu**).
Na dole okna klikamy przycisk **Browse…,** w celu otwarcia okna **Przeglądarka Procedur** (**PP**).
Procedur (z Biblioteki procedur) lub **funkcji** w **PP** (możemy poszukiwać w polu wyszukiwania wg:

W polu wyszukiwania wpisujemy słowo "**image**", jak pokazano poniżej:



Jeżeli nazwa wszystkich wyświetlonych funkcji zaczyna się od słowa "`gimp`" - jest "rodowitą" procedurą GIMP-a. Nazwy zawierają odniesienie do obiektu, z którym funkcja działa.
Większość procedur w oknie *Przeglądarki procedur* ma nazwę odpowiadającą nazwie w GIMP. Przesuwamy suwak w dół, aż po chwili gdy dojdziemy do procedury o nazwie "**gimp-image-new**", czyli tego co jest nam w tej chwili potrzebne.
W prawej części okna, widzimy kilka bardzo ważnych informacji, które musimy znać i stosować.
Jest to szybka informacja o jakiejś funkcji lub poleceniu.
Widzimy, że "**`gimp-image-new`**" tworzy nowy obraz o określonej szerokości, wysokości i typie
(**Np. ..** 400 x 300 RGB, itd.).
W sekcji **Parametry**, widzimy:
width ***INT32*** (szerokość): 1 <= width <= 262144,
height ***INT32*** (wysokość): 1 <= width <= 262144 w pikselach
INT32: oznacza liczbę całkowitą ***integer*** ze znakiem (bez miejsc po przecinku) o długości 32 bitów. Zmienne typu integer nie mogą pamiętać dowolnie dużych liczb całkowitych.
i **typy** obrazów mogą być:
RGB (**0**) – tworzy kolorowy obraz trybu RGB,
GRAY (**1**) – tworzy obraz w gradacji szarości,
INDEXED (**2**) – tworzy obraz indeksowany
To są parametry, których GIMP oczekuje, aby je poprawne podać. Jeśli ich nie podamy lub wpiszemy je nieprawidłowo, dostaniemy zwrócony błąd.
W sekcji **Return Values** *Zwracane wartości* - widzimy, że jeśli podamy poprawne parametry, GIMP zwróci **ID** (identyfikator) nowo tworzonego obrazu (jak podano już powyżej, będzie on tylko w pamięci, do chwili, aż powiadomimy **KPF**, aby nam go pokazała).
Sekcja **Additional Information** Dodatkowe informacje również podaje nam bardzo przydatne informacje.
I nie obejmuje to tylko omawianych tutaj, ale trzeba mieć tego świadomość i czytać dla każdej procedury.

Tworzymy nasz obraz!

Po kliknięciu w Procedure Browser (**PP**) przycisku ***Apply***, lub dwuklikiem na nazwie, wywołana wybrana procedura zostanie wklejona do okna Konsoli jako polecenie Python:

```
image = pdb.gimp_image_new(width, height, type):
```

Widzimy, że **PP** wkleja do **KPF** procedurę już poprawnie opisaną. Nie musimy pamiętać, że trzeba wymienić wyświetlane w oknie łączniki (" **-** ") na podkreślenia (" **_** ").
Teraz musimy wymienić nazwy parametrów (tutaj: "szerokość" , "wysokość" i "typ") na potrzebne nam określone wartości (domyślnie w pixelach):

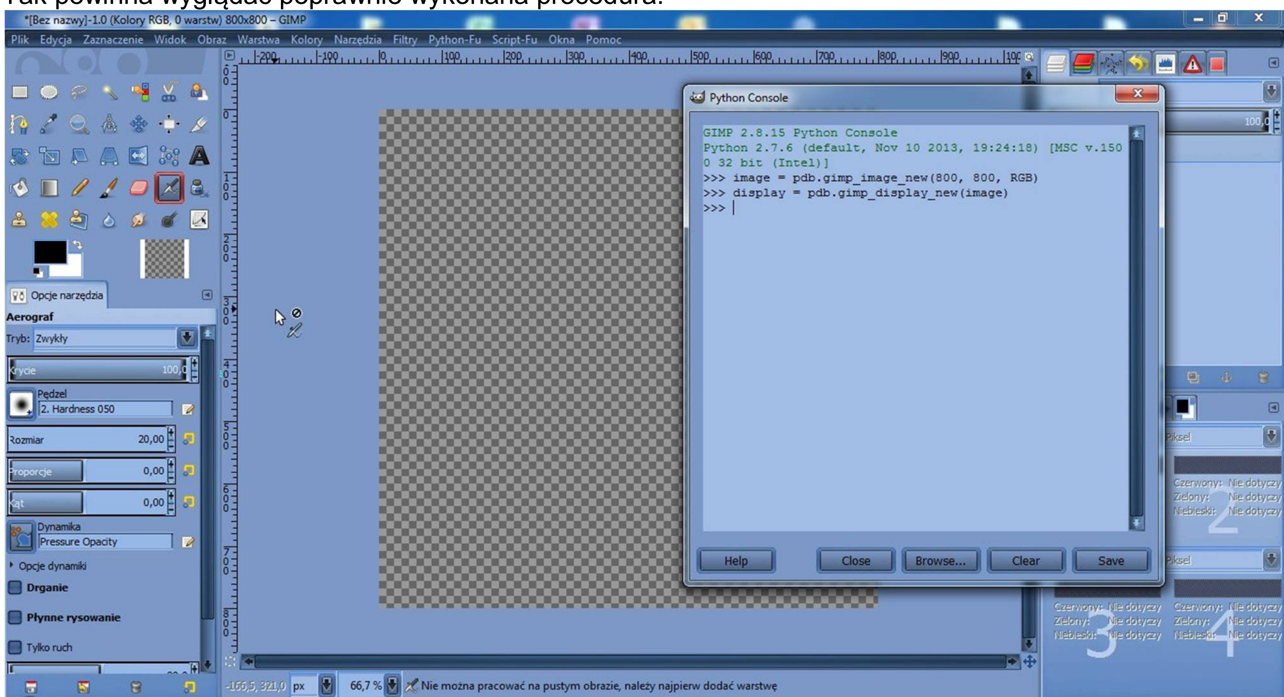
```
image = pdb.gimp_image_new(800, 800, RGB)
```

Po kliknięciu **OK**, zauważymy, że kursor został od razu ustawiony w odpowiednim *następnym* miejscu do wprowadzenia kolejnej procedury (funkcji). Jest to bardzo ładna funkcja!
Ale na ekranie GIMP-a nic się nie pojawiło.

Jeśli po wypełnieniu polecenia, nic nie widzimy, znaczy to, GIMP z powodzeniem utworzył nasz obraz (**ID**) i (przechowuje go w pamięci), teraz możemy uzyskać do niego dostęp za pośrednictwem zmiennej ***image***.
A więc, aby wyświetlić obraz, który stworzyliśmy, musimy w konsoli znowu kliknąć **Browse…,** i w **PP** wpisać **display**. W oknie **PP** *zaznaczamy* **`gimp-display-new`** i klikamy **Apply**.
W oknie **KPF** pojawi się poprawna procedura:

```
display = pdb.gimp_display_new(image)
```

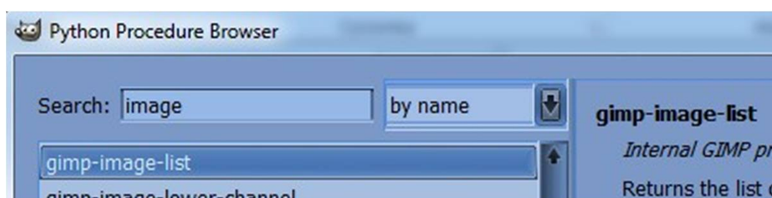
gdzie **image** to nazwa naszego utworzonego wcześniej powyżej obiektu. Trzeba pamiętać o tym, że **image** nie jest domyślnym argumentem tej funkcji (czy raczej procedury, bo nie wymaga ona przypisania do zmiennej).
Gdybyśmy wcześniej utworzony obiekt przypisali do zmiennej o nazwie **obraz**, wywołanie obiektu na ekran wyglądałoby tak: **`display = pdb.gimp_display_new(obraz)`**
Przypomnienie: w przypadku kiedy "*ręcznie*" wpisujemy do **KPF** nazwę procedury, musimy. jak już pisałem, pamiętać, że trzeba zmienić łączniki (" **-** "), na podkreślenia (" **_** ").
Tak powinna wyglądać poprawnie wykonana procedura:



Jeśli chcemy wyświetlić listę wszystkich utworzonych przez Nas obrazów. Wystarczy z **PP** wybrać

```
gimp_image_list(),
```

lub poprawnie wpisać ją "z palca", a pokażą się wszystkie obiekty wraz ze swoimi nazwami.



W naszym przypadku, po kliknięciu **Apply** oczekujemy następującego efektu:

```
Python Console
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> image = pdb.gimp_image_new(800, 800, RGB)
>>> display = pdb.gimp_display_new(image)
>>> num_images, image_ids = pdb.gimp_image_list()
>>> |
```

[**Jeszcze tylko mała wskazówka.**
Podczas "ręcznego" pisania wtyczek, można się zmęczyć cały czas ciągłym pisaniem "**pdb**", więc niektórzy definiują w Pythonie alias "**pdb**" jako "**gimp.pdb.**" i nazywają go np. "**g**".
Potem można pisać takie rzeczy, jak:

```
>>> g = gimp.pdb
>>> g.gimp_edit_fill(target_layer, PATTERN_FILL)
```

To bardzo przydatny skrót, ale tracimy funkcjonalność konsoli Pythona, więc kiedy zaczynamy pisać "`gimp.pdb.gimp_edit`" i naciśniemy dwa razy klawisz **TAB**, konsola pokaże wszystkie możliwe nazwy funkcji, które zaczynają od aktualnie napisanej, jeśli tylko taka istnieje, napisze pełną listę nazw funkcji. Pomocne uzupełnianie nazw.

```
Python Console
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> g = gimp.pdb
>>> gimp.pdb.gimp_edit_
gimp.pdb.gimp_edit_blend(
gimp.pdb.gimp_edit_bucket_fill(
gimp.pdb.gimp_edit_bucket_fill_full(
gimp.pdb.gimp_edit_clear(
gimp.pdb.gimp_edit_copy(
gimp.pdb.gimp_edit_copy_visible(
gimp.pdb.gimp_edit_cut(
gimp.pdb.gimp_edit_fill(
gimp.pdb.gimp_edit_named_copy(
gimp.pdb.gimp_edit_named_copy_visible(
gimp.pdb.gimp_edit_named_cut(
gimp.pdb.gimp_edit_named_paste(
gimp.pdb.gimp_edit_named_paste_as_new(
gimp.pdb.gimp_edit_paste(
gimp.pdb.gimp_edit_paste_as_new(
gimp.pdb.gimp_edit_stroke(
gimp.pdb.gimp_edit_stroke_vectors(
>>> gimp.pdb.gimp_edit_

Help   Close   Browse...   Clear   Save
```
]

Jeśli pragniemy pomocy, ze strony GIMP-a
GIMP **API** (*Application Programming Interface* - Interfejs programistyczny aplikacji*)*, z pomocą Python help()
Jak już pokazano powyżej:

Po otwarciu konsoli wpisujemy:
help(gimp)

lub

import gimpfu
help(gimpfu)

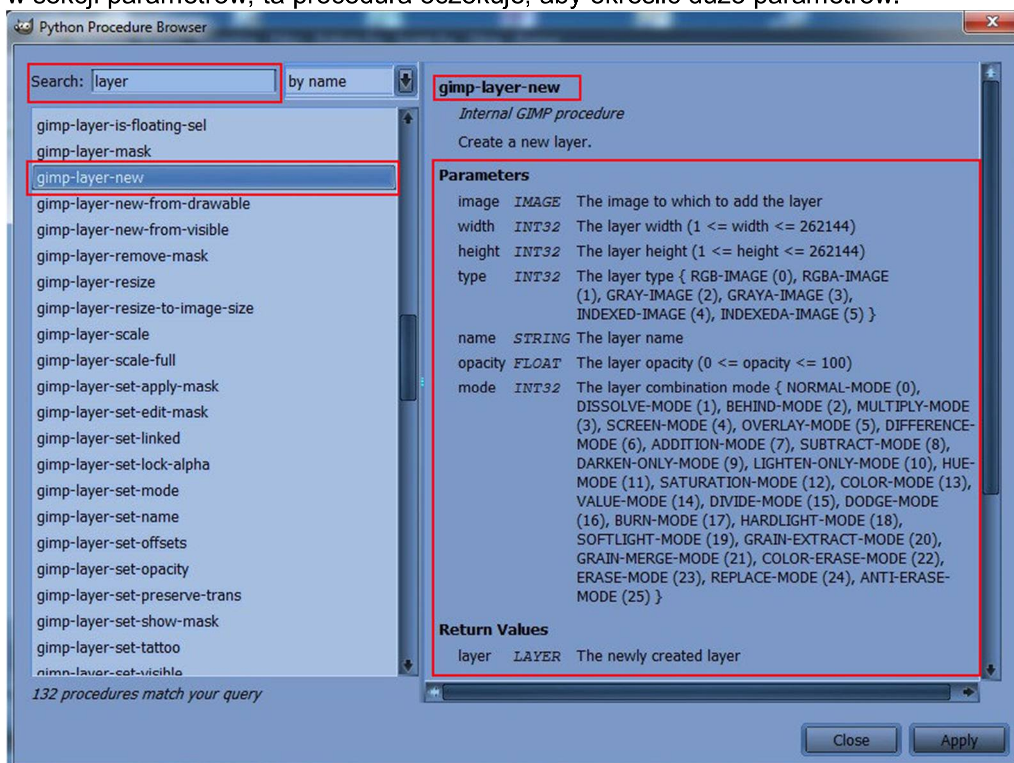
czy nie ciekawe, inne Python GIMP, moduły **(gimpenums i gimpshelf i gimpplugin.)**
gimpenums: użyteczne stałe GIMP-a
gimpshelf: pomaga utrzymać persistent (trwałe) dane plugin (między sesjami GIMP-a.)
gimpplugin: alternatywa dla gimpfu, jeśli potrzebujemy low-level, bardziej szczegółowy dostęp do tworzenia wtyczek, dla wtyczek typu EXTENSION lub TEMPORARY

Większość piszących wtyczki używa **gimpfu**. Importuje gimp i definiuje **alias** (czyli inaczej, znany jako - alternatywna nazwa służąca do identyfikacji obiektów, funkcja polegająca na skróceniu i uproszczeniu nazewnictwa) "`pdb`" do `gimp.pdb`.

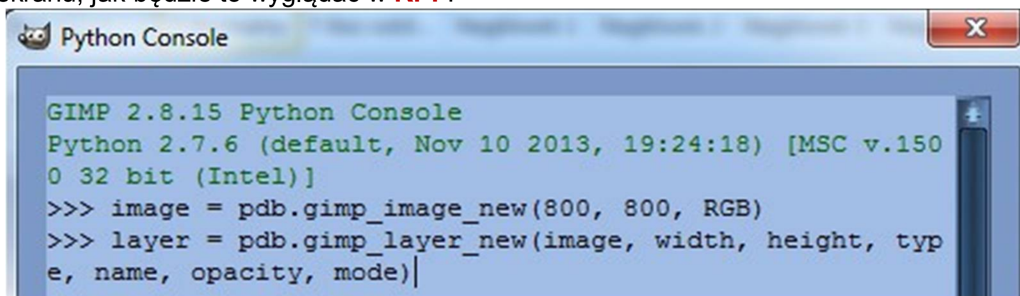
Za pomocą **help(gimp)** jest generowana pomoc (przez program) z samego kodu Pythona (w większości nie edytowana). Część z nich pochodzi z docstrings (komentarze w niektórych miejscach w kodzie Pythona). W docstrings w Pythonie GIMP-a (gimp, gimpfu itp) są niewystarczające lub na przykład, obejmują one licencję, co nie powinno.
Uwaga: nie można skutecznie importować wtyczki i wezwać od niej help().

Tworzymy warstwę

W tym celu wracamy do **PP** i wpisujemy w **Szukaj „layer"**, podobnie jak zrobiliśmy powyżej dla „**image**", przewijamy suwak w dół, aż dojdziemy do "**gimp-layer-new**".
Jak widać, w sekcji parametrów, ta procedura oczekuje, aby określić dużo parametrów.



Podobnie jak to było w przypadku tworzenia obrazu, klikając przycisk "**Apply** Zastosuj" i wklejamy "**gimp-layer-new**" do **KPF**:
Oto zrzut ekranu, jak będzie to wyglądać w **KPF**:

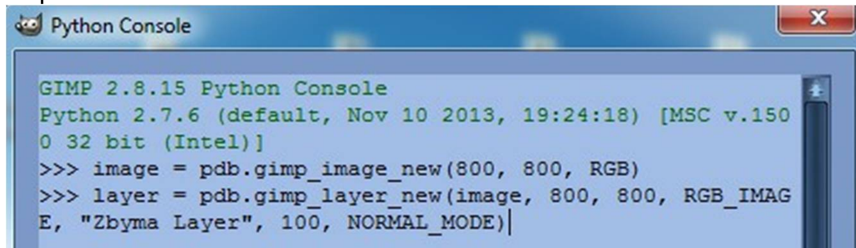


Zrzut pokazuje, że procedura utworzenia warstwy w GIMP, oczekuje od Nas określenia **7** parametrów.
Parametry **kolejno** są następujące:
Image ID - identyfikator obrazu powstał gdy tworzyliśmy obraz w poprzednim punkcie
Width - jaką chcemy szerokość naszej warstwy. Warstwa może być większa lub mniejsza niż wymiary obrazu, ale przyjmiemy, że są równe.
Height - takie same informacje jak dla szerokości.
Type - typ warstwy { RGB-IMAGE (**0**), RGBA-IMAGE (**1**), GRAY-IMAGE (**2**), GRAYA-IMAGE (**3**), INDEXED-IMAGE (**4**), INDEXEDA-IMAGE (**5**) }
Name – nazwa warstwy, którą GIMP doda. Nazwa to łańcuch (ciąg), musimy dołączyć tą informację w cudzysłowie (prostym!).

Opacity – krycie warstwy, przezroczystość warstwy, jako procent. To jest wartość float, dzięki czemu można wprowadzić cyfry z miejscem po przecinku. Waha się od 0% krycia do 100%, a ponieważ chcemy, aby naszą warstwę było widać ustawiamy na 100.

Mode - tryby mieszania warstw mogą być: { NORMAL-MODE (**0**), DISSOLVE-MODE (1), BEHIND-MODE (2), MULTIPLY-MODE (3), SCREEN-MODE (4), OVERLAY-MODE (5), DIFFERENCE-MODE (6), ADDITION-MODE (7), SUBTRACT-MODE (8), DARKEN-ONLY-MODE (9), LIGHTEN-ONLY-MODE (10), HUE-MODE (11), SATURATION-MODE (12), COLOR-MODE (13), VALUE-MODE (14), DIVIDE-MODE (15), DODGE-MODE (16), BURN-MODE (17), HARDLIGHT-MODE (18), SOFTLIGHT-MODE (19), GRAIN-EXTRACT-MODE (20), GRAIN-MERGE-MODE (21), COLOR-ERASE-MODE (22), ERASE-MODE (23), REPLACE-MODE (24), ANTI-ERASE-MODE (25) }.

Przygotujmy sobie wpis:



Pamiętamy:
Do zapisu stałych **Type** i **Mode**: RGBA-IMAGE; NORMAL-MODE; FOREGROUND-FILL; używamy bezwzględnie wielkich liter – *wersaliki*, (separatory rozdzielające ciąg znaków) - trzeba wymienić łączniki ("**-**") na podkreślenia (" _ "): RGB_IMAGE, NORMAL-MODE, nie stosujemy spacji - odstępu pomiędzy dwoma wyrazami.

Ponadto, musimy dać unikalną nazwę warstwy, upewniamy się, że jest ***wpisana w cudzysłowie prostym*** (należy pamiętać, że jest to **łańcuch**).
Należy starannie sprawdzić wpisy przed zatwierdzeniem, inaczej w **KPF** pojawi się komunikat np.:

```
SyntaxError: EOL while scanning string literal,
```

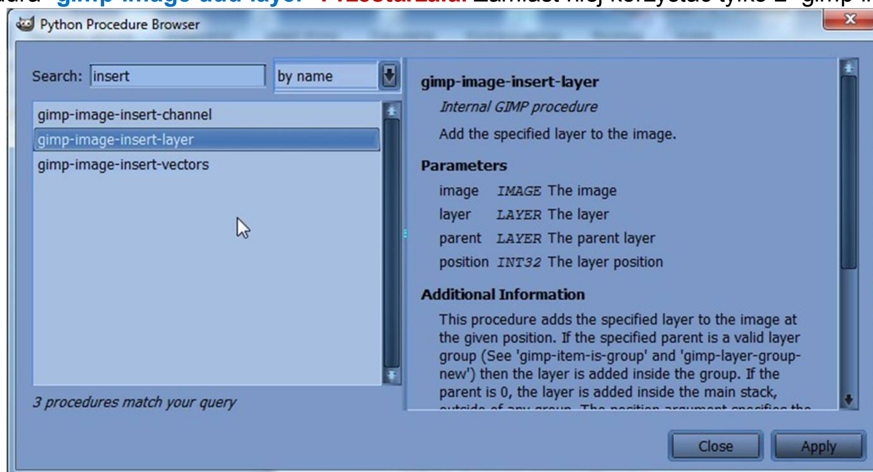
(gdy zapomniałem wstawić ***cudzysłów prosty*** w nazwie warstwy).

Gdy wkleimy parametry, klikamy **Enter**, jeśli zrobimy wszystko poprawnie, w **KPF** nic się nie pojawi, znaczy to że nowa warstwa o nazwie "Zbyma Layer" została utworzona i możemy się do niej zwrócić przez zmienną ***layer***

```
>>> layer = pdb.gimp_layer_new(image, 800, 800, RGB_IMAGE, "Zbyma Layer", 100, NORMAL_MODE)
>>>
```

Dodanie warstwy do obrazu

Stworzyliśmy warstwę, ale ta warstwa istnieje tylko w pamięci GIMP. Musimy dodać ją do obrazu. Wracamy do **PP** i wpisujemy „**insert**" – (dodaj) i wybieramy procedurę "**gimp-image-insert-layer**"
[Uwaga: Procedura "**gimp-image-add-layer**" **Przestarzała.** Zamiast niej korzystać tylko z "gimp-image-insert-layer"]



Dodajemy warstwę do obrazu. W **KPF** zrobimy to przy pomocy procedury:

`gimp_image_insert_layer`

Ta procedura dodaje warstwę do obrazu w określonym położeniu.

Ponownie, w oknie **PP** po prawej stronie widzimy, że teraz procedura wymaga **4** parametrów:
image ID idetyfikator *obrazu*,
layer identyfikator warstwy
parent (***LAYER*** **)** nadrzędna grupa warstw, w której chcemy dodać warstwę. Jeśli **parent** równy jest **None**, warstwa będzie dodana do głównego stosu, poza jaką bądź grupą.
(Zwracam uwagę na to, że: API Python`a oczekuje "*nazwy miasta*" a nie "*kodu pocztowego",* dlatego jeśli zamiast **None** użyjemy **0** – otrzymamy błąd -zrzut poniżej. Nie przyjmuje numerycznych identyfikatorów obiektów, ale referencje obiektów - *odwrotnie jest w script-fu*.)

```
>>> image = pdb.gimp_image_new(800, 800, RGB)
>>> layer = pdb.gimp_layer_new(image, 800, 800, RGB_IMAG
E, "Zbyma Layer", 100, NORMAL_MODE)
>>> pdb.gimp_image_insert_layer(image, layer, 0, 0)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: wrong parameter type
>>> |
```

position pozycja warstwy (**INT32**).
Z podanych w oknie **PP dodatkowych informacji** dowiadujemy się, że:
Argument **position** określa położenie warstwy w podstawowym stosie warstw (lub grupie *jeśli idetyfikator grupy, został dostarczony w argumencie* **parent**).
Czym większa cyfra, tym wyżej zlokalizowana warstawa w stosie. Obliczanie zaczyna się od góry (0) i rośnie. Jeśli **position** równy **-1,** a **parent** jest określony jako **None**, to nowa warstawa będzie wstawiona nad aktywną w danym momencie warstwą (lub wewnątrz grupy, jeśli aktywną jest grupa warstw).
http://developer.gimp.org/api/2.0/libgimp/libgimp-gimpimage.html
http://docs.gimp.org/en/gimp-layer-groups.html
Uwaga: ta procedura ma błędy które mają zostać poprawione od GIMP 2.8 16 wg.:
http://www.widecodes.com/CzVjVqPjUW/gimp-pythonfu-nested-group-layers.html

Ważnym jest przypomnieć o tym, że typ warstwy powinien odpowiadać typowi obrazu bazowego, innymi słowami, nie należy np. próbować w obrazie RGB wstawiać warstwę GRAY.

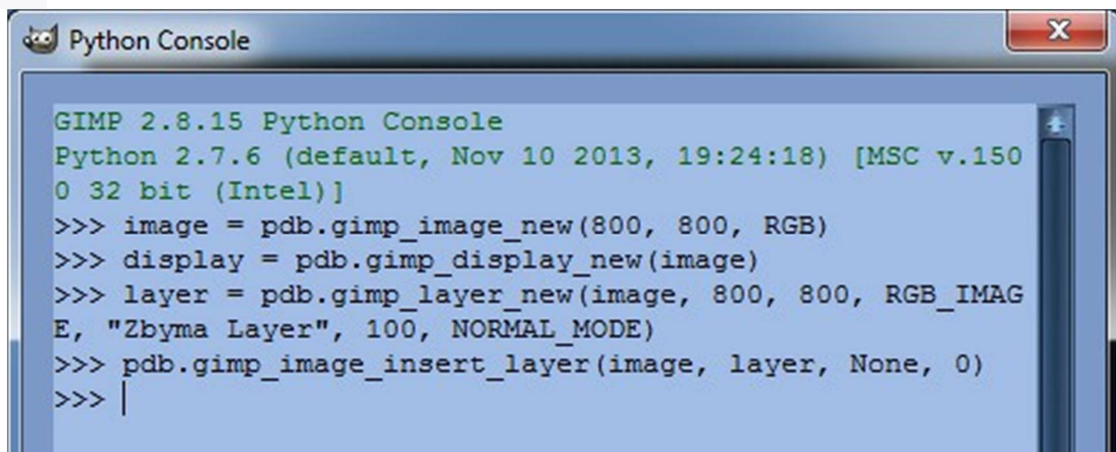
Oprócz tego, *warstwy, nie mające kanału- alfa* (przezroczystości) należy zawsze wstawiać w pozycji 0 (zero).

Tak jak poprzednio, klikamy przycisk "**Apply** *Zastosuj*", aby skopiować i wkleić do **KPF** procedurę:

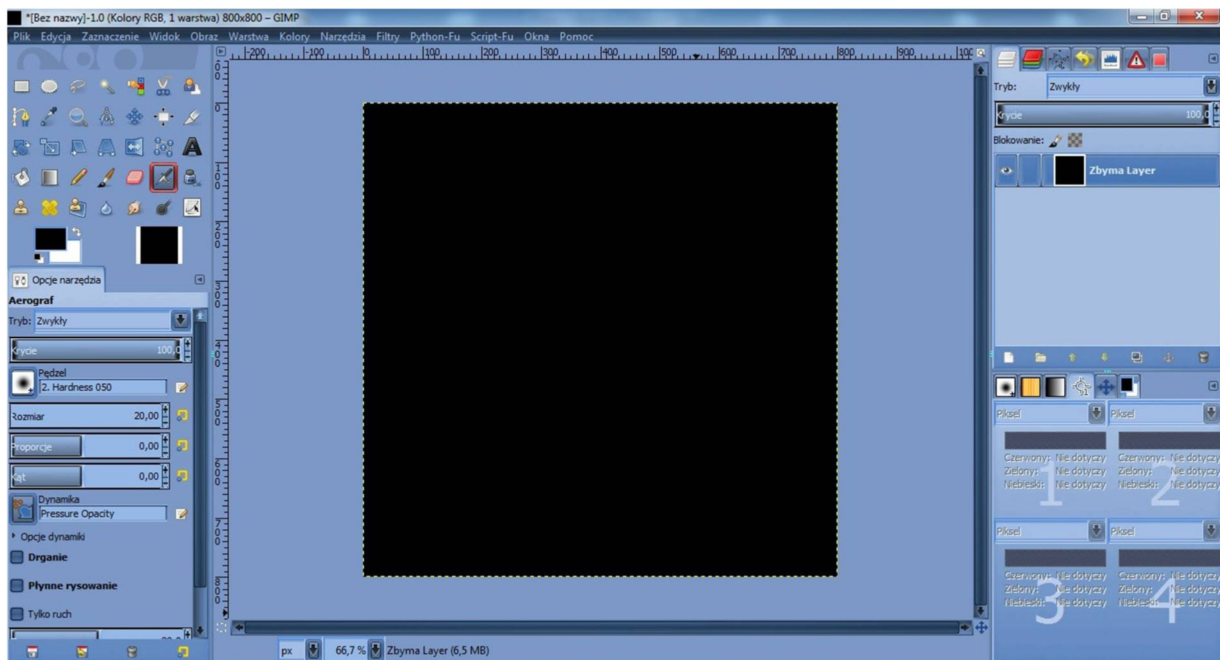
gimp_image_insert_layer

ID obrazu - już istnieje, musimy podać identyfikator warstwy "Zbyma Layer", **parent** None, oraz dodać jej pozycję,
W naszy przypadku, trzeba wbudować warstwę ***layer*** w obraz ***image***, umieszczając ją na dolnym poziomie ponieważ warstwa, nie ma kanału-alfa wstawiamy ją w pozycji 0 (zero) w tym przypadku najwyższa pozycja:

```
>>> pdb.gimp_image_insert_layer(image, layer, None, 0)
>>>
```



Klikamy klawisz **Enter**, jeśli zrobiono wszystko poprawnie, w oknie **KPF** nie powinien pojawić się żaden komunikat o błędzie:
W tym przypadku oznacza to, że warstwa została dodana do naszego obrazu.



Teraz, aby utworzona nowa warstwa uzyskała kolor **Tła** , trzeba warstwą wyczyścić (jeśli tego nie zrobimy, jak widać na zrzucie powyżej, warstawa będzie miała kolor pierwszoplanowy).

Znowu w **KPF** klikamy **Browse…** w celu otwarcia okna **PP,** teraz w w Search wpisujemy **edit**, odszukując procedurę:

gimp-edit-clear

która ma tylko jeden paramer **drawable**
drawable — warstwa jest drawables, czyli obiektem do rysowania.
Więc musimy szukać funkcji służącej do manipulowania warstwą, funkcji, która umożliwia manipulacje na obiekcie - rysowania (na nim).
Co w GIMP-ie jest drawable można sprawdzić:
http://developer.gimp.org/api/2.0/libgimp/libgimp-gimpdrawable.html
Tak więc: warstwy, maski warstw, kanały, selekcje wszystkie są "drawables".
Drawables, jak sama nazwa sugeruje, są rzeczami, które wspierają rysowanie na nich, są obiektami Pixmap.
W zależności od tego, z jakim obiektem pracuje ta czy inna funkcja, w danym przypadku, **gimp-edit-clear** w charakterze parametru należy przekazać wskazanie na warstwę.

Efekt uzyskiwany funkcją **gimp_edit_clear**, zależy od typu warstwy:
Warstawa, zawierająca kanał alfa, w rezultacie czyszczenia stanie się przezroczysta, a warstawa RGB będzie wypełniona kolorem pierwszoplanowym.

Czyścimy warstwę *layer*:

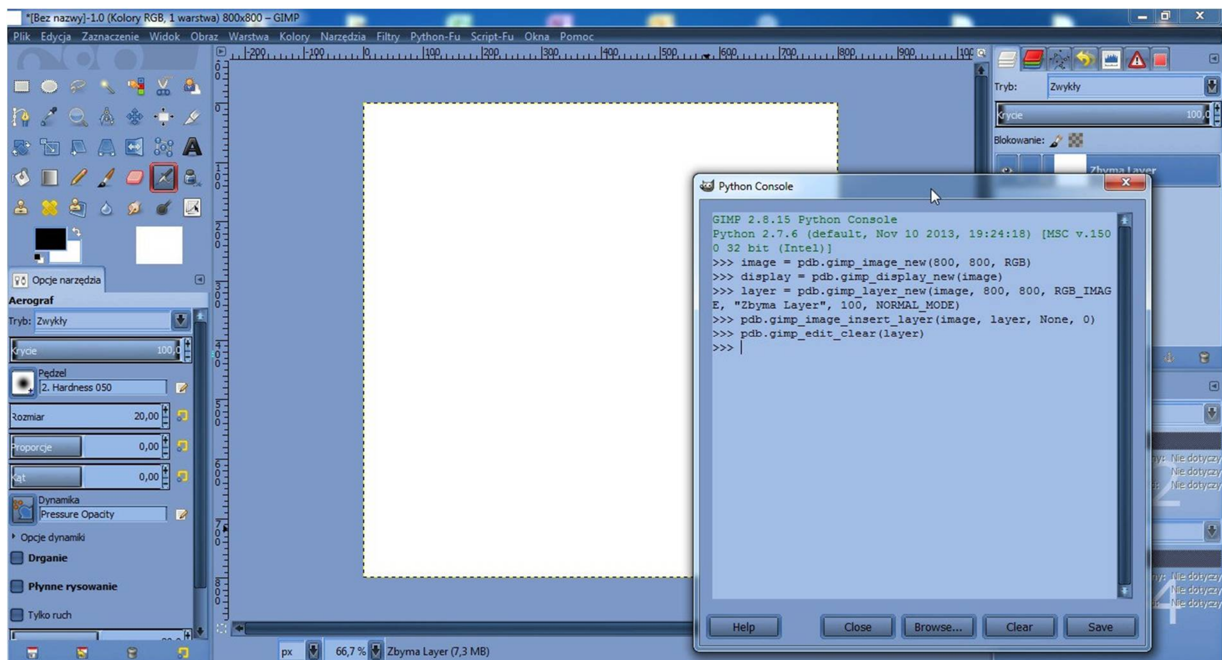
```
>>> pdb.gimp_edit_clear(layer)
>>>
```

Python Console

```
GIMP 2.8.15 Python Console
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.150
0 32 bit (Intel)]
>>> image = pdb.gimp_image_new(800, 800, RGB)
>>> display = pdb.gimp_display_new(image)
>>> layer = pdb.gimp_layer_new(image, 800, 800, RGB_IMAG
E, "Zbyma Layer", 100, NORMAL_MODE)
>>> pdb.gimp_image_insert_layer(image, layer, None, 0)
>>> pdb.gimp_edit_clear(layer)
>>>
```

Nowy obraz jest gotowy, został wywołany na display, poprzednio wpisaną procedurą:

```
display = pdb.gimp_display_new(image)
```



W rezultacie przeprowadzonych powyżej manipulacji, na ekranie pojawiła się zakładka zawierająca obraz RGB o rozmiarze 800x800 pikseli, składający się z jednej warstwy, która jest wypełniona kolorem tła.

W taki oto sposób, potrafimy już sterować GIMP-em za pomocą procedur (funkcji) **API.**

Ale w jaki sposób te manipulacje przetworzyć w formę własnej wtyczki Pytona?
W tym celu, musimy omówić ważne aspekty: rejestrację wtyczki i kod wtyczki o czym będzie poniżej.

«Szkielet» wtyczki, czyli szablon

Aby zacząć pisać Naszą pierwszą wtyczkę, **musimy dokładniej poznać istotną część każdej wtyczki, którą jest funkcja rejestracji wtyczki w PP (PDB).**

Bez tego, GIMP nie będzie ładować wtyczki, ponieważ nie wie, gdzie ją umieścić i jak ją uruchomić.

Funkcja rejestracji oczekuje listy 11 parametrów, które zostaną, kolejno wyszczególnione poniżej:
Szkielet typowej wtyczki, napisany z wykorzystaniem modułów **gimpfu,** wygląda następująco:

```
#!/usr/bin/env python  #informacja dla systemu, oznaczająca, że
w przypadku wykonywania programu powinien on być uruchamiany za
pomocą tego właśnie interpretera, czyli, że musi znaleźć
lokalizację Pythona za pomocą programu env do którego została
podana ścieżka, i dopiero wtedy użyć go jako interpretera.

# -*- coding: utf-8 -*- # Tak definiujemy kodowanie znaków,
#Python musi wiedzieć, że Nasza wtyczka nie jest w ANSI.
# Powyższe dwa kody muszą być zawsze pierwszymi we wtyczce!
from gimpfu import *  # To mówi Pythonowi skąd ma importować
niezbędne moduły. W niektórych wtyczkach spotkamy np.:
gettext.install("gimp20-python", gimp.locale_directory, unicode=True)
- gettext instaluje pliki wielojęzycznych tłumaczeń przygotowane przez
autora, aby wtyczka mogła wyświetlać etykiety nie tylko w języku ang. ale
również w języku lokalnym _()  (szczegóły dalej)
# Teraz deklarujemy funkcję, która będzie wykonywać faktyczne
czynności:
```

```
def plugin_func(image, drawable, args):
  # piszemy dowolną nazwą funkcji o najmniej dwóch parametrach: image i drawable (warstwy)
  # Tutaj kod Naszej wtyczki:
  # każda funkcja będzie z Przeglądarki procedur
  #
```

Wpisujemy w Konsoli import gimpfu => **Enter**, **dalej** **help(gimpfu.register)=> Enter**, zwróci Nam co musimy sprecyzować w **register** (rejestracji wtyczki):

```
>>> import gimpfu
>>> help(gimpfu.register)
Help on function register in module gimpfu:

register(proc_name, blurb, help, author, copyright, dat
e, label, imagetypes, params, results, function, menu=No
ne, domain=None, on_query=None, on_run=None)
    This is called to register a new plug-in.

>>> |
```

Jak widać Parametrami, **register** są:

- **name** - nazwa procedury
- **blurb** - krótki opis wtyczki
- **help** - pomocnicza informacja dokumentacyjna w PDB
- **author**
- **copyright** - prawa autorskie
- **date** – data utworzenia
- **label** – **etykieta**
- **imagetypes** - typ obrazu
- **[] params** - parametry
- **[] results** - wyniki zwracane przez Naszą wtyczkę
- **function** – nazwa funkcji,potrzebnej do przetwarzania, **menupath** – ścieżka do menu
- main – funkcja uruchamiająca wtyczkę

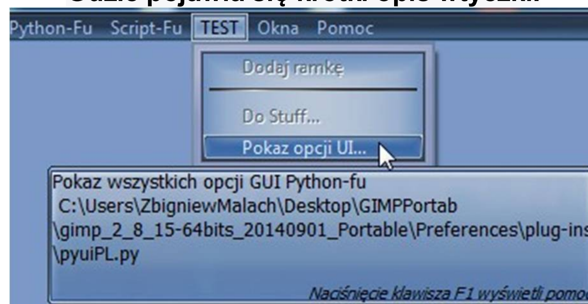
Poniżej omówiono kolejno nazwy (funkcji) wymagane przy rejestracji.
Każdy parametr piszemy w oddzielnej linii
Uwaga: Do rejestracji procedury **Python-fu-register** oprócz obowiązkowych pierwszych 8 powyższych parametrów, potrzebna jest dowolna liczba dodatkowych parametrów, określenie wartości których, jest możliwe w wyświetlonym oknie dialogowym przed wykonaniem skryptu ***param-type*** :
(typ parametru może być np.: **PF_IMAGE, PF_DRAWABLE, PF_COLOR, PF_VALUE, PF_STRING, PF_TOGGLE)**; ***deafult value***, wartości domyślne.
param-type Może praktycznie zawierać, dowolną kombinację elementów kontrolnych, w tym np. do wprowadzania tekstu, wartości numerycznych, przyciski wywołujące dalsze specjalizowane okienka dialogowe wyboru czcionki, koloru, warstw (poziomów), suwaki itp. Każdy element sterujący jest opisany trójką parametrów – typem elementu sterującego (chodzi o stałą np. **PF-...),** dowolnym łańcuchem, który jest wyświetlany w oknie dialogowym (chodzi o opis etykietę - *label*) i trzecim parametrem, którego znaczenie zmienia się w zależności od tego o jaki element sterujący chodzi – może np. chodzić o implementację koloru, nazwę czcionki, nazwę pędzla, zawartość pola listy, itd. Wartości podane przez użytkownika są przekazywane do funkcji przedstawiającej ciało wtyczki jako jej parametry – musimy zwrócić uwagę, aby ilość parametrów odpowiadała ilości elementów w oknie dialogowym a także na ich prawidłową kolejność. Nazwy (stałe) wszystkich typów elementów sterujących, odpowiadają typom danych w **gimpfu**,
Szczegóły omówiono dalej.)

```
register(
        "plugin_func", # Nazwa proc_name rejestrowanej wtyczki, GIMP
zmieni podkreślenia na myślniki i doda prefiks "python-fu" (lepiej
prefiksu samemu nie dodawać) nazwa pojawi się również na krawędzi okna.
        "To jest bardzo funkcjonalna wtyczka" +"\n "+__file__,
```

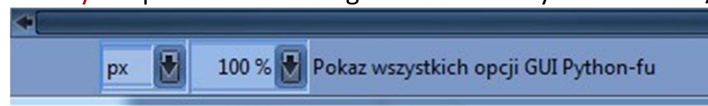
Krótki opis działań wykonywanych przez wtyczkę, zobaczymy go w postaci "*Dymku z podpowiedzią*" **obok kursora myszki** po umieszczeniu kursora na ***Etykiecie*** (nazwie punktu w menu), oraz na **pasku stanu** w dole okna Edytora obrazów i w prawej części okna Przeglądarki procedur a po otwarciu wtyczki jest to tytuł okna GUI.

Można dodatkowo podać (**wg podanego powyżej wzoru**) ścieżkę do katalogu z nazwą wtyczki (**w ścieżce są** ***podwójne*** podkreślenia). Ścieżka pojawia się wtedy obok kursora myszki oraz w oknie wtyczki i w Przeglądarce procedur.
Bardzo pomocne, jeśli chcemy wtyczkę szybko zlokalizować i usunąć, lub przenieść gdzie indziej w menu lub odświeżyć.
Inaczej możemy mieć problem ze znalezieniem wtyczki w menu, gdzie jest wiele elementów i wtyczki mogą być dobrze ukryte, nawet tam gdzie nie podejrzewamy. Nie zawsze jest łatwo znaleźć wtyczkę również w Przeglądarce procedur. Niektóre wtyczki są w kilku wersjach, których różnicą jest często tylko położenie, w różnych miejscach menu.
Dotychczas najlepszym rozwiązaniem było otwarcie wtyczki, poprzez przeciągnięcie jej z folderu *plug-ins* do edytora tekstu i sprawdzenie, gdzie się pojawi.
Lokalizacja w ***register*** jest napisana zawsze w bardzo przejrzysty sposób np. <Image>/ Filters/ Enhance/_Simulate HDR...
Gorzej gdy mamy do czynienia z programowanymi w "C"

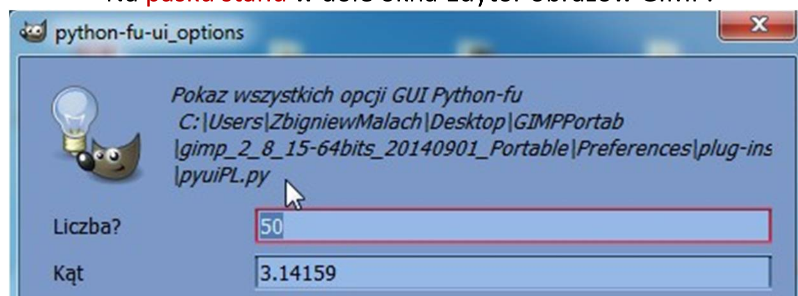
Gdzie pojawia się krótki opis wtyczki.



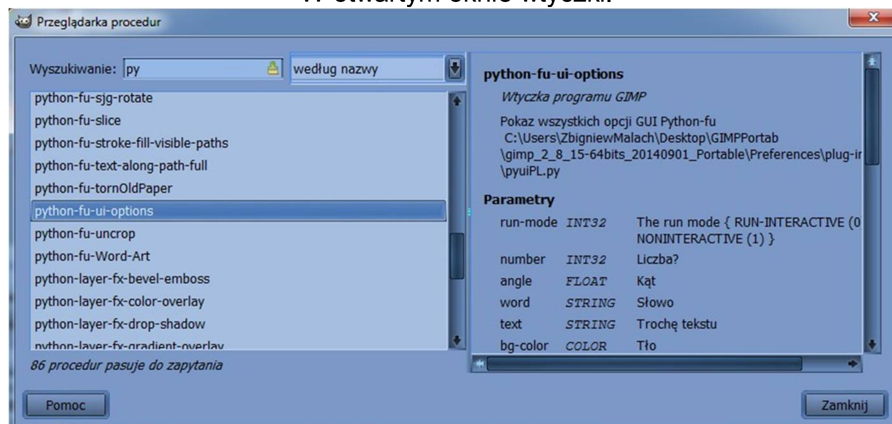
Obok kursora myszki po umieszczeniu go na nazwie wtyczki w menu /podmenu.



Na pasku stanu w dole okna Edytor obrazów GIMP.



W otwartym oknie wtyczki.



W prawej części okna Przeglądarki procedur.

Uwaga: nazwa pliku wtyczki podana w folderze użytkownika, (*Zapisz jako*) nie musi być identyczna z nazwą zarejestrowaną w **PP**, jednak dla łatwego odszukania wtyczki lepiej aby były one identyczne.

```
            "Realizuje to i to, oraz…", # Informacja dokumentacyjna o wtyczce,
która pojawia się w Przeglądarce procedur, należy wyjaśnić w bardziej
szczegółowy sposób funkcje realizowane przez wtyczkę.
            "Autor", # Informacja o Autorze
            "Autor (alfa.beta@gmail.com)", # Informacja o właścicielu praw
autorskich (Copyright) i np. GPL (General Public License) oraz ewent. adres
email
            "30.05.2015", # Data utworzenia wtyczki
            "_RAMKA ...", # Etykieta, nazwa punktu w menu, za pomocą której,
wtyczka będzie uruchamiana. W nazwie Etykiety otwierającej okno dialogowe
wtyczki, często spotykamy wielokropka...(generalnie jest to przestrzegane).
Zastosowanie znaku podkreślenia "_" przed pierwszym znakiem Etykiety spowoduje,
że możemy używać tego znaku jako skrótu klawiaturowego, lepiej wtedy wtyczkę
uruchamiać za pomocą tego skrótu klawiaturowego, etykieta menu nie jest wtedy
tak istotna.
Jest to  jednak wygoda jeżeli w Katalogu/podkatalogu znajduje sie więcej
wtyczek.
        Przykład: http://www.land-of-kain.de/docs/python_gimp/frameedit.py
 Zaznaczamy katalog/podkatalog i kilkamy na literę skrótu.
           "*", # Typ obrazu: jaki typ obrazu obsługuje wtyczka,
"*" oznacz "każdy rodzaj obrazu", Można stosować oznaczenia typów obrazów
"RGB", "RGBA", "GRAY", lub "INDEXED" oznaczeń może być kilka,
oddzielonych przecinkami. Jeśli Nasza wtyczka ma obsługiwać obrazy RGB i
RGBA, piszemy: "RGB*".
Przy stosowaniu dwóch cudzysłowów "", wtyczka tworzy nowy obraz, ale nie
działa z istniejącym.

             [    params
                 (PF_IMAGE, "image", "Obraz wejściowy", None), #
Parametry, które będą  przekazywane wtyczce
                (PF_DRAWABLE, "drawable", "Oryginalna warstwa", None), #
Wszystkie ID dla obrazu, warstwy itd.
                (PF_STRING, "arg", "The argument", "default-value")
            ], # Jak pokazano, pomiędzy tymi dwoma nawiasami klamrowymi,
zawarte są typy parametrów dla procedur (funkcji) PF_* z listą obiektów
(argumentów) – każda taka lista to  krotka.
Krotka parametrów wtyczki:(type, name, description, default, [extra]).
Krotkę można zamknąć w nawias. Krotkę definiuje się przez wpisanie jej
elementów i przedzielenie ich przecinkami. Krotki zazwyczaj używa się w
sytuacjach, gdy w wyrażeniu lub funkcji zdefiniowanej przez użytkownika
można założyć, że zestaw danych nie ulegnie zmianie. Można importować
dowolną stałą PF_* jako typ parametrów. Stosowne elementy interfejsu
użytkownika będą wyświetlane gdy wtyczka będzie wywołana w trybie
interaktywnym. Argumenty pozwalają użytkownikowi na wprowadzanie różnych
rodzajów wartości, jak integer value, tekst, wybór czcionki, zaznaczenie obrazu
lub warstwy, wyboru pliku lub katalogu itd ...
Poniżej będziemy mogli przetestować je wszystkie, mamy wtyczkę z wszystkimi
typami parametrów (wtyczka tworzy bardzo wysokie interakywne okno).
```

Uwaga:

Nawet jeśli zastosujemy pustą `[]`, listę ***params*** (parametrów), wtyczka będzie działać, ale bez otwierania interfejsu (okna) wejściowego, a GIMP automatycznie dodaje **3** parametry

- Aktualny obraz jest 2-gi, 3-ci jest obecny drawable obiekt / warstwa
- Pierwszym parametrem jest "**`run mode`**" tryb uruchamiania – jako:
  " RUN-NONINTERACTIVE" lub "RUN-INTERACTIVE".
  (lub **`run`** z ostatnich wartości), zapewniając **GUI** dla trybu **`interactive`** i zapisywanie ostatnio używanych ustawień).

- jeśli zdefiniujemy pozycję menu **Toolbox**, wymagany jest tylko parametr "**run mode**", co oznacza, że Nasza wtyczka nie będzie wymagała obrazu, ale może generować.

Uwaga: w niektórych wtyczkach możemy spotkać w params zapis:

```
[
(PF_DIRNAME, "indirectory", _("Input Directory"), os.getcwd() ),
(PF_DIRNAME, "outdirectory", _("Output Directory"), os.getcwd() ),
(PF_SPINNER, "new_long_side", _("New long side (px):"), 640, (10, 3000, 1)),
(PF_OPTION, "interpolation", _("Interpolation"), 3, ["None", "Linear", "Cubic",
"Lanczos"]),
],
```

Pojawi się pytanie: jaki jest cel / efekt stosowania powyżej, podkreślenia wiodącego i nawiasów "_()"?
Odpowiedź: Nazwą funkcji może być dowolna kombinacja liter, cyfr i znaku podkreślenia (ale, cyfry nie są akceptowane jako pierwszy znak w nazwie).
Można więc zdefiniować funkcję o nazwie (alias) "_", która wykorzystana jest jako "_()". [Inaczej: aby dany ciąg – **string** - został przetłumaczony, trzeba go otoczyć funkcją **gettext** czyli owinąć w "_()".]
W praktyce jest zazwyczaj definiowana jako funkcja, która znajduje się w przetłumaczonej wersji (zgodnie z wymogami języku użytkownika) na argumencie, jeśli istnieje, a w przeciwnym razie zwraca oryginalny łańcuch. (więcej informacji przykładowo w:
https://pl.wikipedia.org/wiki/GNU_gettext http://docs.php.net/manual/pl/
oraz w Poradniku http://1drv.ms/1j3KbWD)

Słowniczek: ***Drawable*** **– warstwa lub kanał obrazu - kontener warstw, lub innych obiektów jak kanały, ścieżki lub zaznaczenia.**

```
            [], # Spis zmiennych, które zwróci wtyczka – rezultat działania wtyczki
            plugin_func, menu="<Image>/TEST/")
# Nazwa procedury jak wyżej, z def,
# oraz przykładowa ścieżka do punktu menu, w którym będzie umieszczona
```

Etykieta **uruchamiania wtyczki. Opcję menupath powinien zacząć <Image> dla menu lokalnego - wtyczek obrazów, <Toolbox> dla menu głównego - wtyczek narzędzi, czy <Layers>; wszędzie gdzie czujemy miejsce wtyczki.**
A więc możemy użyć także jednego z następujących:
"<Channels>", "<Vectors>", "<Colormap>", "<Load>", "<Save>", "<Brushes>", "<Gradients>", "<Palettes>", "<Patterns>" lub "<Buffers>".
Jeśli wtyczka jest związane z kolorami, próbujemy: "<Image>/Colors/TEST"
Jeśli nie umieścimy przykładowego "/TEST/", to nie tworzy się podmenu, tylko Etykieta będzie bezpośrednio w menu w tym przypadku /Colors/.
Nazwy podane w menupath muszą pozostać w języku angielskim. Np. jeśli po <Image> podamy /Filtry, (a nie /Filters) to na pasku menu zostanie dodana nowa pozycja Filtry.
Pozostałą częścią menupath są prawe ukośniki (slash - syst. Unix) oddzielające kolejne pozycje w ścieżce menu (do Etykiety). Ścieżka może być rozbudowana. Ten parametr jest ważny, jeśli nie podamy go poprawnie, nie będziemy w stanie znaleźć w menu Naszej wtyczki.
Ten **wymóg** określa rodzaju systemu operacyjnego w samej funkcji. Powinniśmy znać, typ systemu operacyjnego w celu ustalenia separatora ścieżki dostępu, "\" lub "/" i sobie z tym poradzić we wtyczce.
Po jakiejkolwiek zmianie w *register* i restarcie GIMP, wtyczka dostaje nowy znacznik czasu, co można sprawdzić w pluginrc. Zawsze jest rezyko, że gdy w istniejącej wtyczce zmienimy dane w *register*, możemy uszkodzić ważne miejsce w kodzie Pythona, wtedy jedyny ratunek usuwamy pluginrc.
Dane rejestracyjne wtyczki zawierają znacznik czasu pliku wtyczki (długi numer, w sekundach), który jest porównywany z aktualnym znacznikiem czasu pliku. Jeśli są one różne, wtyczka zostaje ponownie zarejestrowana, inaczej Gimp pomija rejestrację.
Uwaga:

1. **plik pluginrc może być bezpiecznie usunięty,zostanie automatycznie zregenerowany przez odpytywnie zainstalowanych wtyczek, oczywiście trochę to potrwa gdy dużo wtyczek.**
2. **Nie możemy zmieć pozycji wtyczki w menu, zawsze będą iść do dołu rozwijanej listy danego katalogu/podkatalogu menu.**

Pamiętajmy również:
Jeśli umieścimy "/" (ukośnik) na koniec domyślnego ciągu znaków to wywoła katalog (podkatalog), w którym może być umieszczone kilkanaście etykiet wtyczek tematycznych np. menu="<Image>/TEST/".
Jeśli pominiemy "/" (ukośnik) na końcu domyślnego ciąg znaków to wywoła etykietę wtyczki np. "<Image>/Filters".
W tym miejscu do **register**, dodawanie jest również wywołanie domeny w celu zlokalizowania tłumaczeń wtyczki. Będzie to *krotka*, która składa się z pliku tłumaczeń (bez rozszerzenia **.mo**) i katalogu, w którym zainstalowane są pliki tłumaczeń. Np.:

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

```
        main() # ta funkcja, jest funkcją uruchamiającą wtyczkę.
```

Stałe, symbole gimp, pdb, register i main są importowane i wywoływane poprzez "from gimpfu import *"

Kilka słów dotyczących konwencji nazewnictwa:
Zaleca się stosować w nazwach wtyczek ***name*** małych liter z łącznikami z nazwą funkcji.

`pyton-fu-poradnik-obraz`

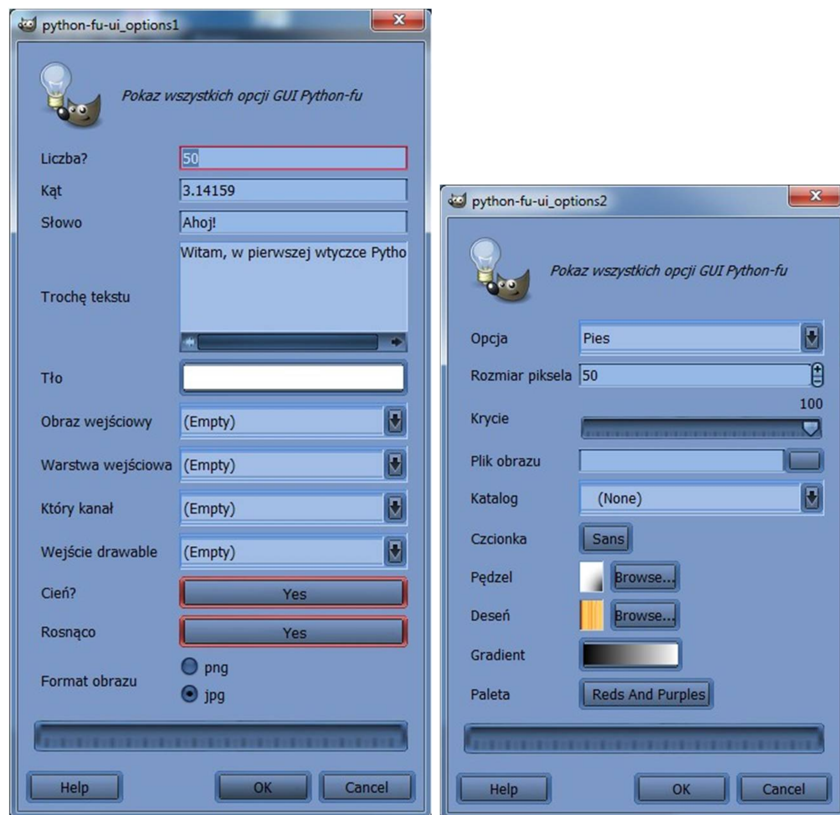
Dobra konwencja podawania nazw wtyczek w GIMP to **python-fu-abc**, bo wtedy wszystkie pojawią się w bazie danych proceduralnych [Przeglądarka procedur (PDB)], wymienione gdzieś w liście **pyton-fu**.
W nazwie rejestrowanej wtyczki, GIMP zmienia podkreślenia na myślniki i dodaje *prefiks* "python-fu" dlatego lepiej prefiksu samemu nie dodawać, nazwa wtyczki pojawi się również na krawędzi okna.
Ale nie wszyscy tego przestrzegają. Pomaga to również potem w odróżnienie ich od plugin-ów.
Trzeba również zwracać uwagę aby nadana nazwa była unikalna dla uruchomienia w PP. Jeśli istnieje inna procedura zarejestrowana w PP, która ma taką samą nazwę, jedna z nich nie będzie działać!.
Sugeruję aby przed wybraniem ostatecznej nazwy procedury, wpisać ją w PP i sprawdzić, czy pojawi się na liście. Jeśli nie, to można np. dodać swoje inicjały na końcu w celu odróżnienia jej od tej drugiej w PP – lub wybierać inną unikalną nazwę. Jeśli jakaś wtyczka nie działa, istnieje duże prawdopodobieństwo (z wyjątkiem błędów we wtyczce), że mamy zarejestrowane w PP dwa egzemplarze procedury o tej samej nazwie.
Przy zapisie gotowej wtyczki część autorów stosuje **Zapisz jako…** odrzucając część pyton-fu- (ja także, ale dopisywałem dla potrzeb tego poradnika **test- => test-sphere32PL** - bo sprawdzano w syst. 32bit).
Niektórzy z autorów zalecają *bardziej opisowe* nazwy dla parametrów i zmiennych, czyli dodanie prefiksu "**in**" do parametrów, aby można było szybko zobaczyć, że są to wartości przekazywane do wtyczki, a nie stworzone w jej obrębie, oraz stosowanie prefiksu "**the**" dla zmiennych zdefiniowanych w wtyczce.

Poniżej,
przykład typów argumentów, które można wykorzystać w bloku rejestracji wtyczki Python-Fu, dla trybu **interaktywnego** (wtedy wtyczka jest do użytku w trybie wsadowym, ponieważ wymaga otwarcia okna deklaracji parametrów), gdzie każdy z nich będzie tworzyć Widget jako element sterowania dialogowego.

Pokazano okno demonstracyjne na bazie którego zaobserwować można zastosowanie wielu typów dostępnych parametrów elementów wejściowych. Ta wtyczka nic nie robi. Po wywołaniu z menu pojawi się **demonstracyjne** okno dialogowe, a w nim są zawarte przykładowe elementy sterujące, wykorzystywane w wtyczkach GIMP-a.

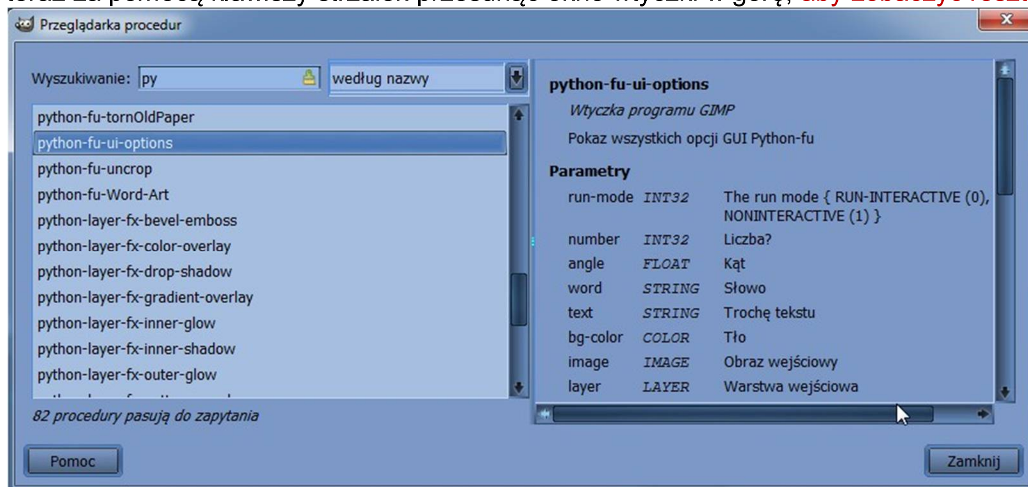
Omówienie okna wtyczki demonstracyjnej

Jeśli chcemy przetestować jak działają poszczególne stosowane elementy sterujące - widżety GUI, **czyli jaka może być struktura (Framework) wtyczki**
wykorzystamy gotową wtyczkę:
(na podstawie Lloyd Konneker 2009r - Python sample scripts oraz
http://gimpbook.com/scripting/ pyui.py: ***by Akkana Peck***)
z wbudowanymi wszystkimi typami widżetów (tworzy bardzo wysokie okno).
Dlatego aby zaprezentować poniżej wygląd stosowanych elementów interfejsu użytkownika, występujących we wtyczce:
http://zbyma.gimpuj.info/pyuiPL.py
podzielono ją (dla zaprezentowania) na dwa pliki, których zrzuty okien są prezentowane poniżej.



Uzyskany Obraz góry i dołu wtyczki demonstracyjnej.

W GIMP (Windows) wystarczy kliknąć prawym przyciskiem myszy na ramce w górze okna wtyczki i kliknąć **Przenieś,** teraz za pomocą klawiszy strzałek przesunąć okno wtyczki w górę, aby zobaczyć resztę okna.



Wygląd w oknie rejestracji procedury pdb.

Po wywołaniu tego okna demonstracyjnego staramy się zrozumieć jak działają poszczególne elementy sterujące - widżet (niektóre z nich omówiono dalej). Zobaczymy, że aktywacja niektórych elementów (np. klikając na nim) wywołuje specjalizowane okna dialogowe GIMP-a, w których łatwo wybrać np. kolor, gradient itd.

Wtyczka, do której link podano powyżej, została przetłumaczona, a jej treść widzimy poniżej.

Listing:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Powyższe dwa kody muszą być pierwszymi we wtyczce
# Pierwszy to informacja dla systemu, oznaczająca,
# że w przypadku wykonywania programu powinien on być uruchamiany za
# pomocą tego właśnie interpretera, czyli, że musi znaleźć lokalizację
# Pythona za pomocą programu env do którego została podana ścieżka,
# i dopiero wtedy użyć go jako interpretera.
# Drugim definiujemy kodowanie znaków, Python musi wiedzieć,
# że Nasza wtyczka nie jest w ANSI.
```

```
# GIMP Python plug-in template showing all UI elements.
# Copyright 2010 Akkana Peck <akkana@shallowsky.com>

# Uwaga: w gimpfu.py jest najbardziej aktualna lista elementów UI
'''
Prezentacja widgets - widżetów, (element interfejsu graficznego, skrót od
"window gadget" dla wtyczek GIMP-a w Python-ie
(w MS Windows używają w tym kontekście terminu "kontrolka" lub "element
kontrolny")

Wtyczka jest szablonem specyfikacji dla rejestracji parametrów,
demonstrującym jak je programować i jak wyglądają w GUI.
Uwagi:
Istnieją podobne typy nazw SF_xxx dla Scheme i języka C.
Przykład publikacja: http://www.gimpuj.info/index.php?topic=53173.0
Podręcznik stylu GUI:
  Nie używamy wielkich liter w pokazanych tutaj opisach, jako dużą piszemy tylko
pierwszą literę.
  Należy używać dwukropka po opisie procedury.
  Wskazane używać skrótów klawiaturowych (podkreślenie przed pierwszą literą
Napisu w ścieżce).
  W nazwie Etykiety otwierającej okno dialogowe, używamy wielokropka ....
  Zalecanym jest przyzwyczajenie się do pisania wyłącznie pojedynczych linii
logicznych w pojedynczych liniach fizycznych.
Tylko w przypadku naprawdę długich linii logicznych możemy je zapisać w kilku
liniach fizycznych (patrz poniżej w def).
  Niektórzy zalecają dokładnie pisać opisy uzasadniające.
'''
# Tłumaczenie i rozszerzenie wtyczki 30.05.2015, Zbigniew Małach

# Poniższe mówi Pythonowi skąd ma załadować moduły GIMP
from gimpfu import *

# To jest procedura, która będzie wykonywać faktyczne czynności

def show_py_ui(number, angle, word, text, bgcolor,
        newimg, newlayer, channel, drawable,
        shadowp, ascendingp, imgfmt, option,
        size, opacity, imagefile, dir,
        font, brush, pattern, gradient, palette):
    return
# Rejestrujemy procedurę w PDB
register(
        "ui_options",  # Jeśli tak zapiszemy nazwę procedury rejestrującej
wtyczkę, GIMP zmieni
              #podkreślenia na myślniki i doda prefiks "python-fu" (samemu
lepiej prefiksu nie dodawać).
        "Pokaz wszystkich opcji GUI Python-fu " +"\n "+__file__,  # Krótki opis
działań wykonywanych
            # przez wtyczkę, zobaczymy go po umieszczeniu wskaźnika myszki na
Etykiecie (nazwie punktu w menu),
            # można ewentualnie dodatkowo podać (wg podanego wzoru) ścieżkę do
katalogu z nazwą wtyczki.
            # Powyższe pojawi się również w Przeglądarce procedur.
      "Nic nie robi, ale wyświetla wbudowane do okna wtyczki elementy interfejsu
graficznego widgets",  # Długa Informacja dokumentacyjna o wtyczce, która pojawi
się w Przeglądarce procedur,
            # należy wyjaśnić w bardziej szczegółowy sposób funkcje realizowane
przez wtyczkę.
        "Nazwisko Autora",
        "Właściciel praw Autorskich + Copyright",  # Informacja o właścicielu
prawach autorskich (Copyright)
            # i np. GPL (General Public License) oraz ewent. adres email
        "2015",  # Data utworzenia wtyczki
```

```
        "_Pokaz opcji UI...",  # Etykieta, nazwa punktu w menu, za pomocą
której, wtyczka będzie uruchamiana,
              # zastosowanie znaku podkreślenia "_" przed pierwszym znakiem
Etykiety spowoduje, że możemy używać
              # tego znaku jako skrótu klawiaturowego, wygoda jeżeli w
Katalogu/podkatalogu znajduje sie więcej wtyczek.
              # Zaznaczamy katalog i kilkamy na literę skrótu.

        "",     # Typ obrazu z którym wtyczka może współpracować, to oznaczenie
dotyczy tworzenia nowego obrazu,
                       # nie działa na istniejącym.
      [
            (PF_INT, "number", "Liczba?", 50),  # Liczby całkowite
            (PF_FLOAT, "angle", "Kąt", 3.14159), # Liczby zmiennoprzecinkowe
            # można także użyć PF_INT8, PF_INT16, PF_INT32
            (PF_STRING, "word", "Słowo", "Ahoj!"),
            # PF_VALUE to inny termin dla PF_STRING
            (PF_TEXT, "text", "Trochę tekstu",
                "Witam, w pierwszej wtyczce Python!"),
            (PF_COLOR, "bg-color", "Tło", (1.0, 1.0, 1.0)),
            # lub można to pisać jako alias PF_COLOUR

            (PF_IMAGE, "image", "Obraz wejściowy", None),
            (PF_LAYER, "layer", "Warstwa wejściowa", None),
            (PF_CHANNEL, "channel", "Który kanał", None),
            (PF_DRAWABLE, "drawable", "Wejście drawable", None),

            (PF_TOGGLE, "shadow", "Cień?", 1),
            (PF_BOOL,   "ascending", "_Rosnąco", True),
            (PF_RADIO, "imagefmt", "Format obrazu", "jpg",
                (("png", "png"), ("jpg", "jpg"))), # Przyciski opcji
            (PF_OPTION, "option", "Opcja", 2, ("Mysz", "Kot", "Pies", "Koń")),

            (PF_SPINNER, "size", "Rozmiar piksela", 50, (1, 8000, 1)),
            (PF_SLIDER, "opacity",  "Krycie", 100, (0, 100, 1)),
            # (PF_ADJUSTMENT jest taki sam jak PF_SPINNER

            (PF_FILE, "imagefile", "Plik obrazu", ""), # Pozwala wybrać
            # plik lub katalog, w zależności od tego, co jest domyślnym ciągiem
            znaków.
            # Jeśli umieścimy '/' (ukośnik) na koniec domyślnego ciągu znaków to
            poprosi
            # o podanie katalogu.
            #Jeśli pominiemy '/' (ukośnik) na końcu domyślnego ciąg znaków
            poprosi o plik.
            (PF_DIRNAME, "dir", "Katalog", "/tmp"),

            (PF_FONT, "font", "Czcionka", "Sans"),
            (PF_BRUSH, "brush", "Pędzel", None),
            (PF_PATTERN, "pattern", "Deseń", None),
            (PF_GRADIENT, "gradient", "Gradient", None),
            (PF_PALETTE, "palette",  "Paleta", ""),

            # (PF_REGION, "region",_("Region"), 100), może działać, ale chyba
            mało użyteczne. Zobacz gimpfu.py.
            # (PF_FILENAME, "filename", _("Nazwa pliku"), ""), #str
            #(PF_VECTORS, "vectors", "Vectors", None),
            #(PF_DISPLAY, "display", "Display", None),
        ],  #Pomiędzy tymi nawiasami klamrowymi zawarte są wyświetlane parametry
GUI w interaktywnym oknie wtyczki
        [], # Spis zmiennych, które zwróci wtyczka - rezultat
            show_py_ui, menu="<Image>/TEST"
# Nazwa procedury (funkcji) podana w def i ścieżka do punktu menu GIMP, gdzie
powinna się znajdować Etykieta uruchamiania okna wtyczki, w nazwie etykiety
```

otwierającej okno wtyczki, używamy wielokropka ..., (ścieżka do menu musi startować z np. <Image> lub <Toolbox> czy <Layers>, wszędzie gdzie czujemy jej miejsce (może zawierać rozbudowaną ścieżkę).

```
        )
main()  # Uruchamia wtyczkę
```

Po rejestracji jednej lub więcej funkcji wtyczki, *Musimy* wywołać funkcję **main**.

```
gimp.main(init_func, quit_func, query_func, run_func)
```

Spowoduje to rozpoczęcie pracy wtyczki. Gdy są potrzebne **GUI** będą wyświetlane, a funkcja Naszej wtyczki zostanie wywołana w odpowiednim czasie.

Wyjaśnienia dotyczące niektórych Widget:

Widget - podstawowy element GUI interfejsu graficznego, skrót od "window gadget" np. okno, pole edycji - wyboru, suwak, przycisk. (w MS Windows używają w tym kontekście terminu "kontrolka" lub "element kontrolny"). Przydatne we wtyczkach w trybie interaktywnym.

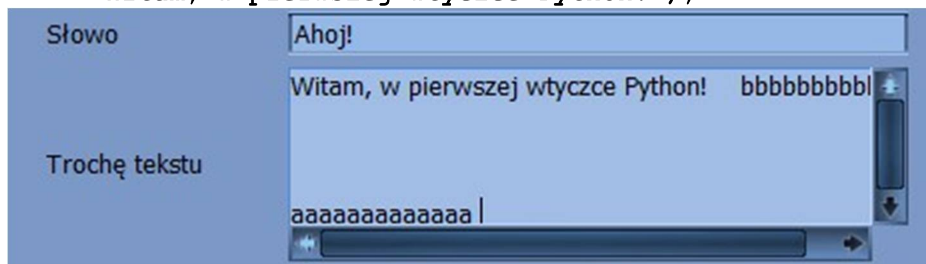
```
(PF_IMAGE, "image", "Obraz wejściowy", None),
```

Przydatne tylko w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który jest rozwijanym polem wyboru, który pozwala użytkownikowi wybrać otwarty obraz.

```
(PF_DRAWABLE, "drawable", "Input drawable", None),
```

Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który jest rozwijanym polem wyboru, który pozwala użytkownikowi wybrać `DRAWABLE` (np. warstwę **Tło**) z otwartego pliku obrazu.

```
(PF_STRING, "word", "Słowo", "Ahoj!"),
# PF_VALUE to inny termin dla PF_STRING
(PF_TEXT, "text", "Trochę tekstu",
                "Witam, w pierwszej wtyczce Python!"),
```



Text i Multi-line text

PF_TEXT podobny do **PF_STRING** i **PF_VALUE**, ale w polu można tworzyć więcej niż jeden wiersz tekstu. widget wpisu jest znacznie większy i może mieć zewnętrzne suwaki.
Przydatne w trybie interaktywnym.
Jeśli tekst, przekracza granicę *szerokości* pola wprowadzania, lub dodano ilość wierszy przekraczającą *wysokość* pola wprowadzania, automatycznie jest dodawany suwak poziomy lub pionowy, ewentualnie obydwa.
W celu wyświetlania całej treści, wewnątrz łańcucha na końcu linii (wiersza), dodajemy znak nowej linii (**\n**). Można dodać znak np. podwójnego odstępu (nowej linii) **\n\n** (*escape* **ucieczka).**
Wartość zwracana PF_TEXT , gdy skrypt zostanie wywołany, łańcuch zawierający cały wprowadzony tekst.

```
(PF_COLOR, "bg-color", "Tło", (1.0, 1.0, 1.0)), # (Red, Green, Blue)
            # lub można to pisać jako alias PF_COLOUR
```

Akceptuje argumenty kolorów. W oknach dialogowych, podgląd kolorów, otworzy się okno wyboru koloru który po kliknięciu będzie tworzony, Po kliknięciu otworzyśię okno "Wybór koloru Python-Fu."
Określanie barw w różnych formatach, zawiera od 3 do 4 wartości:

```
$color = canonicalize_colour ("#ff00bb"); # html format
$color = canonicalize_colour ([125,128,0]); # RGB
$color = canonicalize_colour ([255,255,34,255]); # RGBA
$color = canonicalize_colour ([1.0,1.0,0.32]); # actual GimpRGB
$color = canonicalize_colour ('red'); # uses the color database
```

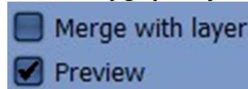
```
(PF_IMAGE, "image", "Obraz wejściowy", None),
(PF_LAYER, "layer", "Warstwa wejściowa", None),
(PF_CHANNEL, "channel", "Który kanał", None),
```

```
            (PF_DRAWABLE, "drawable", "Wejście drawable", None),# obraz, kanał
lub warstwa
```

PF_TOOGLE lub PF_BOOL Wartość logiczna Wyświetla pole, aby zaznaczyć/odznaczyć wartość logiczną Prawda lub Fałsz. (TRUE 1 lub FALSE 0), wykorzystywany dla przycisku przełączania w etykiecie!

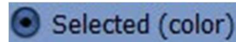
```
            (PF_TOGGLE, "shadow", "Cień?", 1), # A boolean value
            (PF_BOOL,   "ascending", "_Rosnąco", True),# A boolean value
```

"Kanon" GUI, mówi, że przyciskiem **tak / nie** powinien być checkbox.- przycisk wyboru (w działaniu bardzo podobny do przycisku przełączania), jeśli chodzi o wygląd to jest to zwykle mały prostokącik,



który w zależności od stanu albo jest "czysty", albo jest na nim narysowany znaczek "tick" (**v** - z wydłużonym prawym ramieniem) jak widać ☐ lub ☑

```
(PF_RADIO, "imagefmt", "Format obrazu", "jpg",
                (("png", "png"), ("jpg", "jpg"))), # Przyciski opcji
```

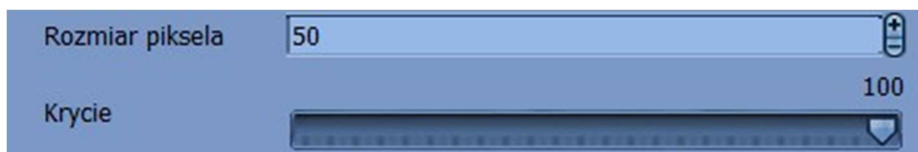



```
(PF_OPTION, "option", "Opcja", 2, ("Mysz", "Kot", "Pies", "Koń")),
```

Najpierw specyfikujemy parametry stałe "**PF-constans**". Powodem, aby korzystać z **extra** stałych jest to, że dają one **gimpfu** więcej informacji, więc może on tworzyć lepszy interfejs, (przykładowo typ **PF_FONT** jest równoważny typowi **PARAM_STRING**, ale w **PF_FONT** mamy w **GUI** przycisk, który powoduje pojawienie się okna wyboru czcionki).
Typy **PF_SLIDER, PF_SPINNER i PF_ADJUSTMENT** spodziewają się pięciu elementów, czyli wymagają dodatkowych parametrów. Są one w postaci (**min, max, step**), i dają limity na przyciski przełączania krokowego lub suwak.
Jeśli jest tylko jedna wartość zwrotu, funkcja wtyczki powinien zwrócić tylko tę wartość. Jeśli jest więcej niż jedna, należy funkcji wtyczki zwrócić krotkę wyników.

```
(PF_SPINNER, "size", "Rozmiar piksela", 50, (1, 8000, 1)),
(PF_SLIDER, "opacity",  "Krycie", 100, (0, 100, 1)),
           # (PF_ADJUSTMENT jest taki sam jak PF_SPINNER
```

Tworzy widget sterowania dialogowego w formie pola tekstowego lub suwaka:



Slider – suwak,
Spinner, SpinButton lub roll-box składa się z pola tekstowego oraz strzałek w górę i dół z prawej strony. pozwala użytkownikowi na wybór wartości z zakresu wartości numerycznych. Wartość może mieć regulowaną liczbę miejsc po przecinku, a wielkość kroku jest konfigurowalna. Wybór jednego z przycisków powoduje, zmianę wartość w górę i w dół z zakres możliwych wartości. Pole wpisu może również być edytowane bezpośrednio do określonej wartości.

```
Usage:
   PF_SLIDER "label" '(value lower upper step_inc page_inc digits type)
```

Dostosowanie parametrów wymaga 7 wartości argumentów okna.
Kolejno poniżej:
lista argumentów Okna (Widget) - podstawowy element GUI.
"Label" – Etykieta, czyli tekst pokazany z przodu: pole edycji, suwak, przycisk
value – Domyślna wartość startowa

`lower / upper` – Min-value max-value dolna i górna wartość (zakres regulacji).
`step_inc` – Przyrost / zmniejszenie wartości.
`page_inc` – Przyrost / zmniejszenie wartości za pomocą klawiszy.
`digits` – Cyfry po przecinku (dziesiętne części). [int=0 lub float=1]
`type` – **PF_SLIDER** lub **0**, **PF_SPINNER** lub **1**

Przykładowa lista:

```
(PF_SPINER, "Rozmiar", 400,(50, 1000, 1, 20 0 1))
```

400 – reprezentuje na liście parametrów, domyślną wartością startową. Oczywiście potem możemy zastosować, jakąś inną ale w granicach podanych poniżej.
50 - reprezentuje najmniejszą wartość, od jakiej możemy rozpocząć (wpisana w pole).
1000 - jest to największa wartość, jaką użytkownik może wprowadzić.
1 - jest to przyrost krokowy, czyli o ile **spinner** przesunie się po każdym kliknięciu. Pisząc wtyczkę mamy zestaw możliwości, aby przejść w krokach co 1, ale można ustawić na 10, 25, 50, itd. Wybór należy do Nas.
20 - jest to przyrost o ile przesunie się **slider - suwak** z każdym kliknięciem. Jeśli klikniemy obok ślizgacza suwaka, będzie się poruszał w odstępach określonych powyżej.
Przy pisaniu wtyczki wartości używamy według własnego uznania.
`digits` wartość ta może być tylko **0** (zero) lub **1** (jeden).
`Default = 0` - wskazuje GIMP, że wartości wprowadzone przez użytkownika będą liczbami całkowitymi (**`PDB_INT`** cyfry bez miejsc po przecinku).
(Uzaadnienie: czy ma sens ustawiać spiner co 0,5)
Jedynka wskazuje GIMP, że wartości wpisane przez użytkownika mogą mieć miejsca po przecinku **`PDB_FLOAT`**. Przykładowo, aby określić ustawienie, rozmycia Gaussa, można użyć liczb dziesiętnych.
Czyli typ liczb zależy do projektanta wtyczki, i dlatego musimy wiedzieć, jakie liczby mogą być użyte.
`type`
Zero wskazuje użycie suwaka, jedynka wskazuje GIMP, że chcemy używać spinner (spinner w literaturze opisowej widet, można spotkać się również z określeniami roll-box lub spinbutton).
Do brakujących wpisów zostają podstawione wartości domyślne.

```
(PF_FILE, "imagefile", "Plik obrazu", ""),
```

Pozwala wybrać plik lub katalog, w zależności od tego, co jest domyślnym ciągiem znaków.
Jeśli umieścimy '/' (ukośnik) na koniec domyślnego ciągu znaków to poprosi o podanie katalogu.
Jeśli pominiemy '/' (ukośnik) na końcu domyślnego ciąg znaków poprosi o plik.

```
(PF_DIRNAME, "dir", "Katalog", "/tmp"),
```

Przydatne w trybie interaktywnym. Bardzo podobne do PF_FILENAME, ale tworzony widet, sterowania dialogowego, który jest rozwijanym polem wyboru wskazującym Katalog (*Folder*). Jeśli przycisk zostanie naciśnięty pojawi się możliwość wyboru przez użytkownika innego katalogu.
Zastosowanie:
`PF_DIRNAME "Katalog"` `można stosować sposób wskazania drogi` **`"C:/"`** `lub` **`"C:\\"`**
`PF_DIRNAME "label" /home/katalog1/katalog2/ (UNIX)`
Windows ma inny sposób na wskazanie drogi (używa \ , ukośnik wsteczny – backslash, zamiast / , ukośnika - forward slash)
Wartość zwracana, gdy skrypt zostanie wywołany łańcuch zawierający dirname nazwę katalogu.

Uwaga:
Windows posługuje się konwencją nazewnictwa katalogów **UNC** (ang. *Universal Naming Convention*).
Zgodnie z nią do identyfikowania nazwy komputera stosowany jest *podwójny* ukośnik *wsteczny* **** (ang. backslash), natomiast pojedynczy *ukośnik wsteczny* **** wskazuje katalog na dysku tego komputera np. "\\?\C:\katalog1\katalog2" co zostanie zinterpretowane jako "C:\katalog1\katalog2".

```
(PF_FONT, "font", "Czcionka", "Sans"),
```

Tworzy widget sterowania dialogowego, jako przycisk z etykietą "..." domyślnie ustalonej czcionki.
Po kliknięciu przycisku pojawi się wyskakujące okno, w którym możemy wybierać dowolne czcinki i każda z cech może być modyfikowana.
Zwraca fontname jako ciąg znaków.
Rozmiar podany w fontname jest ignorowany. Jest wykorzystywany tylko w font-selektor.

lista argumentów Spinbutton

"Label" – Etykieta, tekst opisu podanego na przycisku.

"Fontname" – Nazwa domyślnej czcionki.

(PF_BRUSH, "brush", "Pędzel", None),

Przydaje się w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który obejmuje obszar podglądu oraz przycisk, po naciśnięciu którego, uzyskamy podgląd w postaci wyskakującego okna, w którym pędzle mogą być wybierane i każda z cech pędzla może być modyfikowana.
Rzeczywista wartość zwracana, gdy wtyczka zostanie wywołana, lista składająca się z nazwy pędzla, odstępów, krycia, i trybu pędzla w tych samych jednostkach jaką wartość domyślną podano.
Zastosowanie:

PF_BRUSH "Brush" '("Circle (03)" 100 44 0)

"BRUSH" – etykieta, czyli tekst, który będzie pokazany przed polem podglądu

Tutaj dialog pędzla pojawi się komunikat z pędzlem domyślnie Circle (03) odstęp 44, krycie 100 i tryb Zwykły Normal (wartość 0). Jeżeli tych opcji nie zmienimy wartość przekazywana do funkcji jako parametr zostanie ("Circle (03)"100 44 0).

(PF_PATTERN, "pattern", "Deseń", None),

Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który składa się z obszaru podglądu wybranego desenia i przycisku (po kliknięciu którego pojawi się wyskakujące okno "Wybór desenia"). Pozwala nam wybrać inny deseń.
Zastosowanie:

PF_PATTERN "Pattern" "Pine"

"Pattern" - etykieta, czyli tekst, który będzie pokazany przed polem podglądu
"Pine" – deseń wybrany pierwotnie w wtyczce.
Wartość zwracana, gdy wtyczka zostanie wywołana łańcuch zawierający wzorzec nazwy. Jeżeli powyższego wyboru nie zmieniono to łańcuch będzie zawierał "Pine".

(PF_GRADIENT, "gradient", "Gradient", None),

Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który jest przyciskiem zawierającym podgląd wybranego gradientu. Jeśli przycisk zostanie wciśnięty pojawi się wyskakujące okno wyboru gradientu. Pozwala nam wybrać inny gradient.
Zastosowanie:

PF_GRADIENT "Gradient" nazwa gradientu np. "Yellow Orange"

"Gradient" - etykieta, czyli tekst, który będzie pokazany przed polem podglądu
"Yellow Orange" – gradient wybrany pierwotnie w wtyczce.

(PF_PALETTE, "palette", "Paleta", ""),

Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który składa się z przycisku z "..." etykietą (po kliknięciu którego pojawi się wyskakujące okno "Wybór palety"). Pozwala nam wybrać inną paletę.
Zastosowanie:

PF_PALETTE "Palette" "Gold "

"Palette" - etykieta, czyli tekst, który będzie pokazany przed przyciskiem
"Gold" – deseń wybrany pierwotnie we wtyczce.
Wartość zwracana, gdy wtyczka zostanie wywołana, łańcuch zawierający wzorzec nazwy. Jeżeli powyższego wyboru nie zmieniono łańcuch będzie zawierać "Gold".

(PF_FILENAME, "filename", _("Nazwa pliku"), ""), #str => ciąg znaków => napis

Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który składa się z przycisku, *zawierającego nazwę pliku*. Jeśli przycisk zostanie naciśnięty pojawi się wyskakujące okno "Wybór pliku", gdzie użytkownik może zmienić plik związany z tym przyciskiem.
Zastosowanie:

PF_FILENAME "Nazwa pliku" "/itd" czyli np.: **"D:/Pictures/Do prob/P7236541.JPG"**

"***.roz**" wyrażenie typu dowolna_nazwa_pliku (maska).rozszerzenie
Wartość zwracana, gdy wtyczka zostanie wywołana, łańcuch zawierający nazwę pliku.

~/home/katalog1/katalog2/plik.jpg (UNIX)
http://www.gimpusers.com/forums/gimp-developer/13796-python-fu-difference-between-pf_file-and-pf_filename-and-how-to-ask-for-a-file-name-to-create
From: Ofnuts: różnica między PF_FILE i PF_FILENAME i jak prosić o nazwę pliku, aby *create*

PF_FILENAME: Pozwala użytkownikowi wybrać tylko plik.
PF_DIRNAME: Pozwala użytkownikowi wybrać tylko katalog.

PF_FILE: Pozwala użytkownikowi wybrać plik lub katalog, w zależności od tego, co jest domyślnym ciągiem znaków.
Jeśli umieścimy '/' (ukośnik) na koniec domyślnego ciągu znaków, to poprosi użytkownika o podanie katalogu.
Jeśli pominiemy "/" (ukośnik) na końcu domyślnego ciąg znaków poprosi użytkownika o plik.

Przykłady:
(**PF_FILE**, "pf_afile imagefile", _("**Choose File:**Wybierz plik:"), "/**home** Użytkownicy/Zbyszek"), # Wybierz plik
(**PF_FILE**, "pf_adir dir", _("**Choose Directory:**Wybierz Katalog:"), "/**home** Użytkownicy/Zbyszek/"), # Wybierz Katalog
lub
(**PF_FILE**, "pf_afile imagefile", _("**Choose File:**Wybierz plik:**"**), **""**),
(**PF_FILE**, "pf_adir dir", _(**"Choose Directory:**Wybierz Katalog: **"**), **"/"**),

(**PF_FILE**, "pf_afile imagefile", "Plik obrazu", ""),
(**PF_DIRNAME**, "pf_adir dir", "Katalog", "/tmp"),
https://www.mail-archive.com/gimp-web-list@gnome.org/msg00082.html !!!
Skrypt musi być świadomy rodzaju systemie operacyjnym hosta, a dokładniej, czy separator pliku jest "\" lub "/"
==

PF_ENUM
Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, którym jest pole wyboru, wewnątrz którego zawierają się linijki z określonym tekstem do wyboru. W polu pokazują się wszystkie wartości enum dla danego typu enum. Musi to być nazwa zarejestrowanego enum (na przodzie bez prefix "GIMP"). Drugi parametr określa wartość domyślną (przy użyciu wartości enum nicku).
Zastosowanie:
PF"ENUM "Interpolation" '("InterpolationType" "linear")
PF"ENUM "Interpolation (Scaling)" '("InterpolationType" "lanczos")
Wartość zwracana, gdy skrypt zostanie wywołany odpowiada wartości wybranego enum.

PF_CUSTOM
PF_CUSTOM jest dla potrzebujących, nietypowych-widżetów.
Musimy dostarczyć odniesienie kodu zwracającego trzy wartości jako dodatkowy argument: (**widget, settor, gettor**)
<**widget**> jest widget Gtk, który powinien być zastosowany.
<**settor**> to funkcja, która ma jeden argument, nową wartość dla widżetu (widget powinien zostać odpowiednio zaktualizowany).
<**gettor**> to funkcja, która powinna zwrócić wartość bieżącą widgetu. Podczas gdy wartości mogą być dowolnego typu (tak długo, jak to pasuje do skalara), powinniśmy być przygotowani, aby uzyskać ciąg, gdy wtyczka jest uruchamiana z linii poleceń lub poprzez PDB.
Wspomnieć należy również o:
Parasites są to po prostu wrappers – (osłony, opakowania) zawierające
gimp.Parasite(name, data, flag)
Tworzy nowy obiekt parasites, który może być dołączony do **image** lub **drawable**.
Parasites sa obiektami danych zawierającymi:

```
name
data         a string
flags
is_persistent
is_undoable
```

zestaw funkcji, które są używane do manipulowania parasites. Występują one jako funkcje w **`gimp module metods`** dla **`image`** i **`drawable`**. Szczegóły: => http://www.gimp.org/docs/python/
Parasite działa wewnątrz innej aplikacji i pozwala na manipulowanie aplikacją. Stąd sądzić należy, że nazwa jest odpowiednia - pasożyty.
" Patrz dalej: http://shallowsky.com/blog/gimp/gimp-export-scaled.html użyłem GIMP parasites: małe kawałki dowolnych danych można dołączyć do każdego obrazu. Znam temat parasites, długo, ale nigdy nie miałem okazji, aby je wcześniej wykorzystać w plug-in Python. Jestem zadowolony że są one udokumentowane w oficjalnej dokumentacji GIMP Python gimp.docs.python i pracowały jak udokumentowane.
Łatwo było je przetestować, za pomocą: konsoli Pythona (Filtry => Python-fu => Konsola ...), coś podobnego typu:

```
img = gimp_image_list()[0]
img.parasite_list()
img.parasite_find(img.parasite_list()[0]
```

"
dalsze info:
http://developer.gimp.org/api/2.0/libgimpbase/libgimpbase-gimpparasite.html#GimpParasite
http://www.efalk.org/Docs/Python/gimp_0.html#gimp_d_parasite_find
http://chipx86.github.io/gtkparasite/
http://gimp.1065349.n5.nabble.com/Using-parasites-in-script-fu-td25473.html

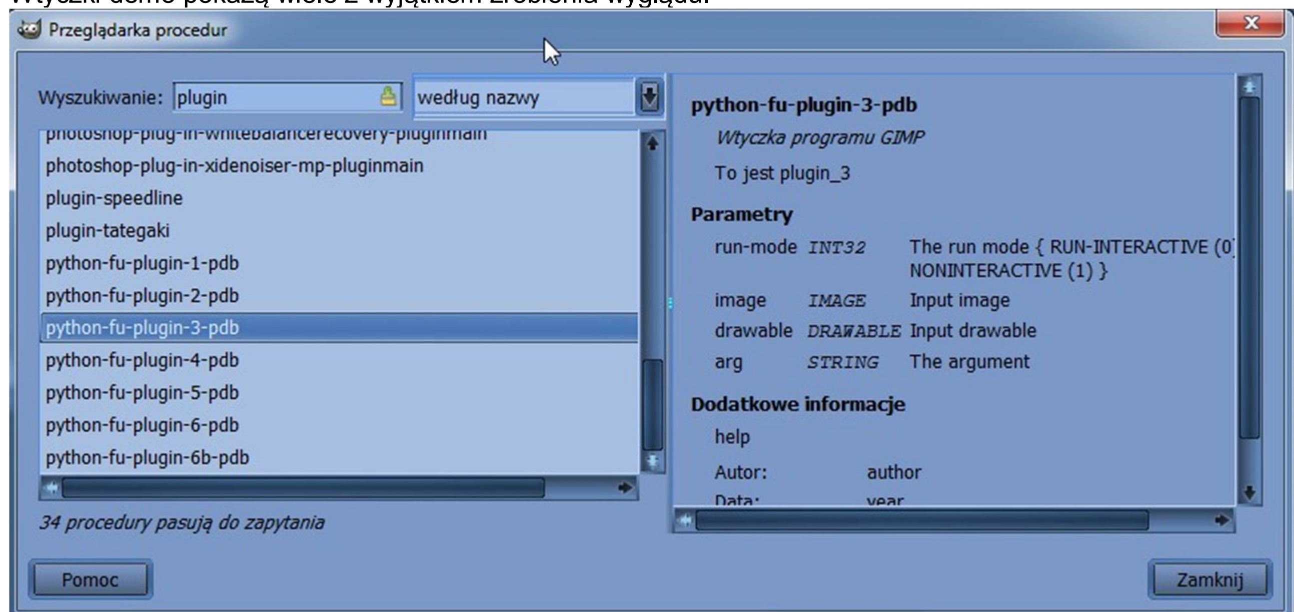
==

Parametry funkcji są "nośnikami" informacji powierzanych im przez użytkownika w oknie dialogowym, po uruchomieniu interaktywnej wtyczki.

Funkcja **`register()`** może być wywołana na kilka różnych sposobów jak to pokazano poniżym szablonie:

Przykłady szablonów rejestracji wtyczek z użyciem gimpfu, dla programistów

http://registry.gimp.org/node/24279
plugin-examples.py.txt April 7, 2010 — bootchk
http://zbyma.gimpuj.info/plugin-examplesPL.py zmieniony i przetłumaczony **PL**
Ten plik zawiera przykłady lub szablony rejestracji wtyczek za pomocą `gimpfu.register()`, omówione powyżej.
Jest to przeznaczone dla początkujących programistów Pythona GIMP, a nie użytkowników.
Kiedy ten plik zainstalujemy, umieszcza on siedem wtyczek demo w GIMP-ie (**plugin_1** do **plugin_6a**).
Wtyczki demo pokażą wiele z wyjątkiem zrobienia wyglądu.



Pokazują one różne kombinacje:
- gdzie pojawi się (lub nie) wtyczka w menu (lub gdzie indziej) w GIMP-ie
- gdy elementy menu wtyczki są włączone
- jakie parametry wtyczka przyjmuje do **`plugin_main ()`** gimp.main ()
- czy lub kiedy wtyczka otwiera okno GUI dla użytkownika, aby mógł wybrać opcje (ustawiane parametry.)

Więcej informacji uzyskamy otwierając plik wtyczki np. w **Notepad++.**

Bieżąca instalacja GIMP może uruchamiać skrypty Pythona, jeśli mamy wpis "Filtry => Python-fu", a także jeśli mamy np. "Filtry => Renderowanie => Chmury => Fog ..." (standardowy filtr napisany w Pythonie, tylko, który będzie pracować i pokazać się, jeśli wersja Pythona jest **OK**). Jeśli w kompilacji nie był Python 2.6 lub wyżej (teraz 2.7.6) - które są obecnie instalowane wraz z wersją GIMP 2.8 Windows – nie będzie.
Nie trzeba szukać dalej, czy Gimp obsługuje Pythona.
Można uzyskać wiele wskazówek, rozpoczynając od zaznajomienia z kolekcją przykładowych wtyczek opracowanych przez **ofnuts:**
Narzędzia GIMP Python **http://sourceforge.net/projects/gimp-tools/files/scripts/**
oraz **http://sourceforge.net/projects/gimp-path-tools/files/scripts/** .
To co one robią i jak one działają jest opisane w:
http://gimp-tools.sourceforge.net/
oraz http://gimp-path-tools.sourceforge.net/
Z pytaniami można zwrócić się do Autora.
Spróbuj od otworcia prostej wtyczki, takiej jak "clear-layers-0.2.py" zarejestruj ją i uruchom.

Długo trwało, omówienie minimum niezbędnych wiadomości, ale ostatecznie wiemy już jak określone manipulacje przetworzyć w formę własnej wtyczki Pytona.

Czyli, dalej już mniej słów, więcej kodu:

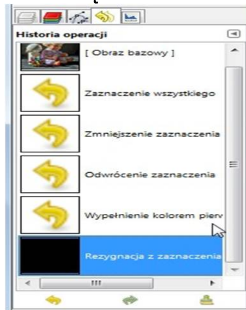
Przechodzimy od słów do dzieła.
Mamy ostatecznie napisaną prostą demonstracyjną wtyczkę, która będzie rysować wokół obrazu ramkę o zadanym rozmiarze i kolorze. W tym celu wykorzystamy podany powyżej szkielet wtyczki i wpiszemy do niego niezbędny kod:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Importujemy niezbędne moduły API GIMP
from gimpfu import *
def add_colored_border(image, drawable, border_width,
border_color):
  pdb.gimp_context_push()
  # Zakazujemy zapisu informacji dla umżliwienia cofnięć,
  # aby wszystkie operacje wykonane wtyczką można było anulować
# jednym klikiem, klikając Ctrl + Z lub wybierając w menu
# "Edycja" punkt "Cofnij".
  pdb.gimp_image_undo_group_start(image)
# Powiększamy rozmiar płótna i przesuwamy na wymaganą odległość
  pdb.gimp_image_resize(image,
                              pdb.gimp_image_width(image)  +
border_width,
                              pdb.gimp_image_height(image)  +
border_width,
                              border_width / 2,
                              border_width / 2)
  # Zapamiętujemy kolor Tła.
  # Jest to również zasada dobrego tonu:
  # pozostawiać po sobie stan początkowy
  old_background = pdb.gimp_context_get_background()
  # Zmieniamy kolor Tła
  pdb.gimp_context_set_background(border_color)
  # Spłaszczamy obraz
  pdb.gimp_image_flatten(image)
  # Przywracamy do stanu początkowego kolor Tła
```

```
  pdb.gimp_context_set_background(old_background)
  # Aktualizujemy, odświerzamy przetwarzany obraz na ekranie po
zmianach
  pdb.gimp_displays_flush()
# Zezwalamy na zapis informacji dla zmian działania, koniec
bloku cofnięć (od początku procedury - funkcji)
  pdb.gimp_image_undo_group_end(image)
  pdb.gimp_context_pop()

# Rejestrujemy funkcję w PDB
register(
          "add_colored_border", # Jeśli tak zapiszemy nazwę
procedury rejestrującej wtyczkę, GIMP zmieni podkreślenia na
myślniki i doda prefiks "python-fu".
          "Dodanie kolorowej ramki do obrazu", # Krótki opis
# działań wykonywanych przez wtyczkę, zobaczymy go po
# umieszczeniu wskaźnika myszki na nazwie punktu w menu.
          "Rysuje dookoła obrazu ramkę podanego koloru i
podanej szerokości ", # Informacja o wtyczce,
# która pojawia się w Przeglądarce procedur
          "Alfa Beta", # Autor wtyczki
          "Alfa Beta (alfa.beta@gmail.com)", # Informacja o
właścicielu prawach autorskich (Copyright) i np. GPL (General
Public License) oraz ewent. adres email
          "30.05.2015", # Data utworzenia wtyczki
          "Dodaj ramkę", # Etykieta, nazwa punktu w menu, za
# pomocą której, wtyczka będzie uruchamiana
          "*", # Typ obrazu z którym wtyczka może pracować
          [
              (PF_IMAGE, "image", "Obraz wejściowy", None),
# Wskazanie aktywnego obrazu
              (PF_DRAWABLE, "drawable", "Oryginalna warstwa",
None), # Wskazanie aktywnej warstwy
              (PF_INT, "border_width", "Szerokość ramki",
"10"), # szerokość ramki
              (PF_COLOR, "border_color",  "Kolor ramki",
(251,3,20)) # Kolor ramki, określamy go w postaci listy 3
wartości RGB, zastosowano kolor czerwony, wartości odczytano z
okna "Zmiana aktywnego koloru".

          ], # Pomiędzy tymi nawiasami klamrowymi zawarte są
parametry wyświetlane w interaktywnym oknie wtyczki
          [], # Spis zmiennych, które zwróci wtyczka rezultat
          add_colored_border, menu="<Image>/TEST/") # Jak
wyżej, nazwa procedury (funkcji) i ścieżka do punktu menu, w
którym umieszczone będzie uruchamianie wtyczki

main() # funkcja uruchamiająca wtyczkę
```

Jak powyżej widać możemy stosować dwa polecenia, które działają w tandemie: **(gimp-image-undo-group-start)** jako początek procedury, która pozwoli nam anulować wszystkie działania naszej wtyczki w GIMP-ie jako jedna operacja, czyli anulować działanie całej wtyczki, a nie tylko ostatnie działanie wtyczki. Musimy pamiętać o dodaniu funkcji zakończenia wywołanej procedury **(gimp-image-undo-grupa-end)**. Ostateczny cel zostanie osiągnięty przy użyciu kombinacji klawiszy (Ctrl + Z) => lub **Cofnij**.

Jest to właściwe w przypadku, jeżeli przez użycie wtyczki obraz został już dopracowany lub wtyczka powinna być uruchomiona z innymi parametrami.
W trakcie testowania kodu wtyczki, można stosować czasowe wyłączanie powyższych dwóch procedur.
Wystarczy wtedy umieścić przed procedurą znak komentarza czyli "#" na czas trwania testów.
Możemy wtedy korzystać z historii wszystkich cofnięć.



Polecenie **`(gimp-displays-flush)`** stosujemy, aby odświeżyć przetwarzany obraz na ekranie.
Ta procedura ma zastosowanie tylko we wtyczkach z otwartymi komponentami obrazu.
Dla wtyczek, które generują nowy obraz, powinna być stosowane procedura **`(gimp-display-new)`**.

Dalej:
Kopiujemy napisaną powyżej wtyczkę **Ctrl+C**, otwieramy **Notepad++** => **Plik** => **Nowy** i wklejamy **Ctrl+V**
Zalecane ustawienie w preferencjach programu, by każdy nowy dokument był tworzony w formacie:
Koduj UTF-8 bez BOM. Ustawić również, by kodowanie to automatycznie zostało ustawione, dla nowo otwieranych plików z formatowaniem **ANSI.** Oto nasz wynik:

```
#!/usr/bin/env python  #
# Poniżej definiujemy kodowanie znaków, Python musi wiedzieć,
#że Nasza wtyczka nie jest w ANSI.
# -*- coding: utf-8 -*-
# Poniżej mówimy Python skąd ma importować niezbędne moduły
from gimpfu import *
def add_colored_border(image, drawable, border_width, border_color):
  pdb.gimp_context_push()
  # Zakazujemy zapisu informacji dla umżliwienia cofnięć,
  # aby wszystkie operacje wykonane wtyczką można było anulować jednym klikiem
  # klikając Ctrl + Z lub wybierając w menu "Edycja" punkt "Cofnij"
  pdb.gimp_image_undo_group_start(image)
# Powiększamy rozmiar płótna I przesuwamy na wymaganą odległość
  pdb.gimp_image_resize(image,
                    pdb.gimp_image_width(image)  + border_width,
                    pdb.gimp_image_height(image)  + border_width,
                    border_width / 2,
                    border_width / 2)
  # Zapamiętujemy kolor Tła.
  # Jest to również zasada dobrego tonu:
  # pozostawiać po sobie stan początkowy
  old_background = pdb.gimp_context_get_background()
  # Zmieniamy kolor Tła
  pdb.gimp_context_set_background(border_color)
  # Spłaszczamy obraz
  pdb.gimp_image_flatten(image)
  # Przywracamy do stanu początkowego kolor Tła
  pdb.gimp_context_set_background(old_background)
  # Aktualizujemy obraz na wyświetlaczu
  pdb.gimp_displays_flush()
# Zezwalamy na zapis informacji dla zmiany działania
  pdb.gimp_image_undo_group_end(image)
  pdb.gimp_context_pop()
# Rejestrujemy funkcję w PDB
register(
        "python-fu-add-colored-border", # Nazwa rejestrowanej funkcji
        "Dodanie kolorowej ramki do obrazu", # Krótki opis działań wykonywanych przez wtyczkę,
        # zobaczymy go po umieszczeniu wskaźnika myszki na nazwie punktu w menu.
        "Rysuje dookoła obrazu ramkę podanego koloru i podanej szerokości ", # Informacja o wtyczce,
        # która pojawia się w Przeglądarce procedur
        "Alfa Beta", # Autor wtyczki
        "Alfa Beta (alfa.beta@gmail.com)", # Właściciel praw autorskich (Copyright)
        "30.05.2015", # Data wykonania
        "Dodaj ramkę", # Etykieta, nazwa punktu w menu, za pomocą której, wtyczka będzie uruchamiana
        "*", # Typ obrazu z którym wtyczka może pracować
        [
          (PF_IMAGE, "image", "Obraz wejściowy", None), # Wskazanie obrazu
          (PF_DRAWABLE, "drawable", "Oryginalna warstwa", None), # Wskazanie warstwy
          (PF_INT, "border_width", "Szerokość ramki", "10"), # szerokość ramki
          (PF_COLOR, "border_color",  "Kolor ramki", (251,3,20)) # Kolor ramki, potrzebujemy pojedynczej
          # wartości, która składa się z listy 3 wartości RGB, przyjąłem kolor czerwony, wartości odczytano z
          # okna Zmiana aktywnego koloru.

        ], # Pomiędzy tymi nawiasami klamrowymi zawarte są parametry wyświetlane w interaktywnym oknie wtyczki
        [], # Spis zmiennych, które zwróci wtyczka rezultat
        add_colored_border, menu="<Image>/TEST/") # Nazwa oryginalnej funkcji i punkt menu, w którym umieszczone
        # będzie uruchamianie wtyczki

main() # Funkja uruchamiająca wtyczkę

```

Po zapisaniu **Plik => Zapisz** np. na Pulpicie pojawi się:

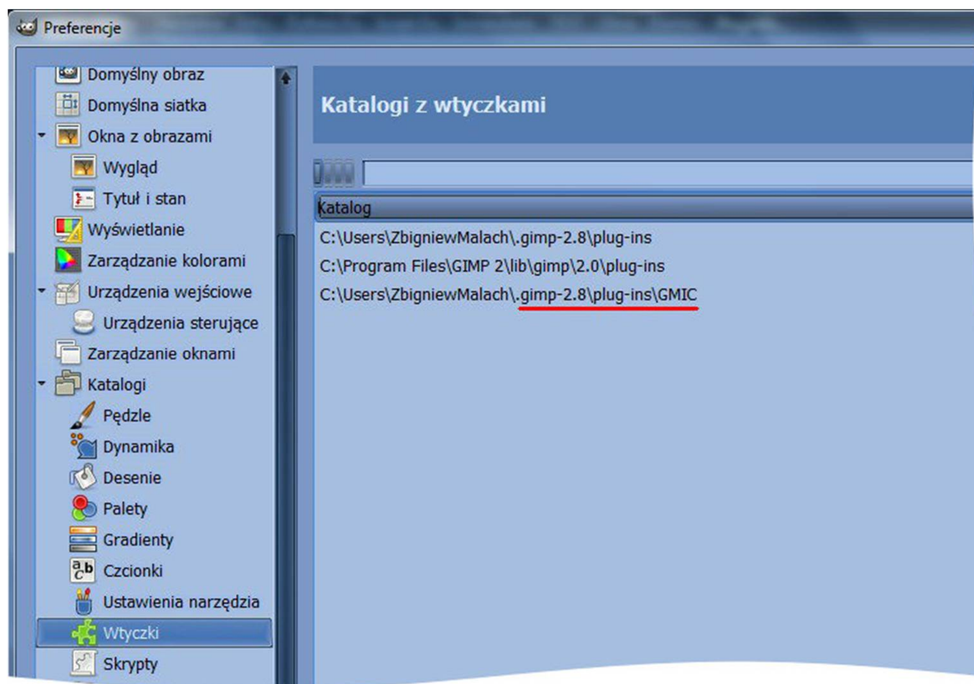
python-fu-add-colored-border.py

Gdzie instalujemy gotowe wtyczki?

Własne napisane wtyczki w Pythonie lub ściągane pliki wtyczek (plug-in) z rozszerzeniem **.py**
np. z: http://registry.gimp.org/taxonomy/term/52 zalecane jest przechowywać lokalnie, zapisując je w folderze użytkownika, który można znaleźć w:

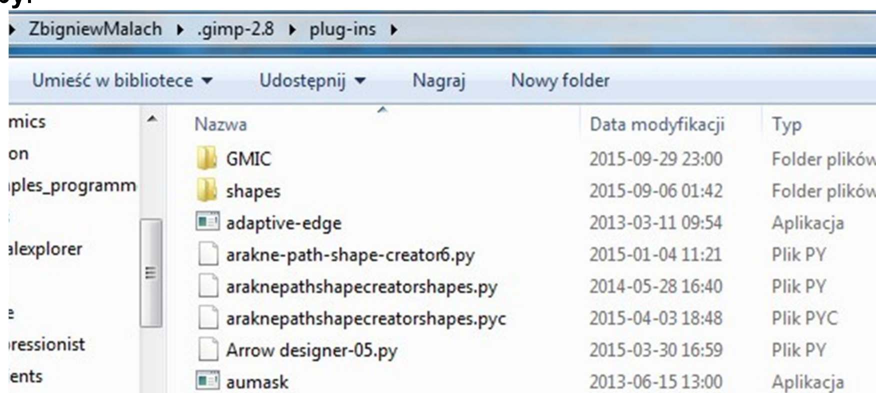
Windows: "C:\Użytkownicy\Użytkownik\.gimp-2.x\plug-ins\" , i w Linux: "~/.gimp-2.x/plug-ins/".

Własne wtyczki Pythona (lub ściągane) możemy zapisać wszędzie, a następnie po otwarciu w menu **Gimp => Preferencje**, **Katalogi**, możemy znaleźć **Wtyczki** i wskazać tu ścieżkę do podkatalogu z zapisaną wtyczką. Z zrzutu listy widać że utworzono dodatkowy katalog z **GMIC** (dlaczego, wyjaśniono w związanym Poradniku).



Gdy uruchomimy GIMP-a ponownie, wtedy zmiana jest brana pod uwagę. Lista folderów pokazuje także, co jest dla całego systemu przechowywane dla wszystkich użytkowników w folderze wtyczek (\gimp\)

Dla przykładu, że mogą tam zostać zapisane wtyczki napisane w języku Python i zostaną one zauważone przez program GIMP, zrzut z ekranu pokazuje ścieżkę docelową do danego folderu oraz kilka wtyczek z rozszerzeniem **.py**.



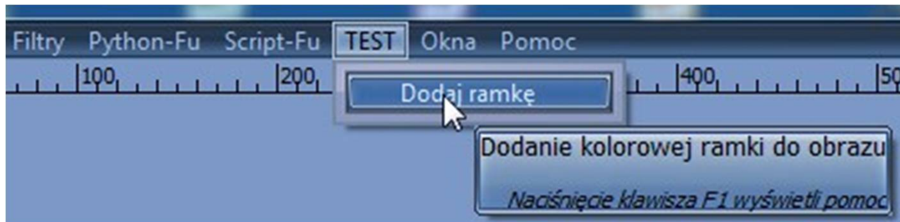
Najlepszym sposobem sprawdzenia czy dana wtyczka, została poprawnie zainstalowana, jest przejście do miejsca interfejsu programu GIMP,w którym wtyczka ma być dostępna, co podano powyżej (natomiast w wtyczkach "ściągniętych" można wyczytać gdzie mają się zainstalować, otwierając je np. w **Notepad++**). Warto tu również dodać, że wtyczka Python-Fu pod Linuxem, musi mieć uprawnienia ustawione na "wykonywalny".

Uruchamiamy GIMP-a.
Jeśli w oknie GIMP-a na pasku menu, pojawiła się pozycja **TEST**, Nasza wtyczka została zarejestrowana poprawnie , z tego miejsca po kliknięciu Etykity "Dodaj ramkę" będziemy uruchamiać Naszą wtyczkę.

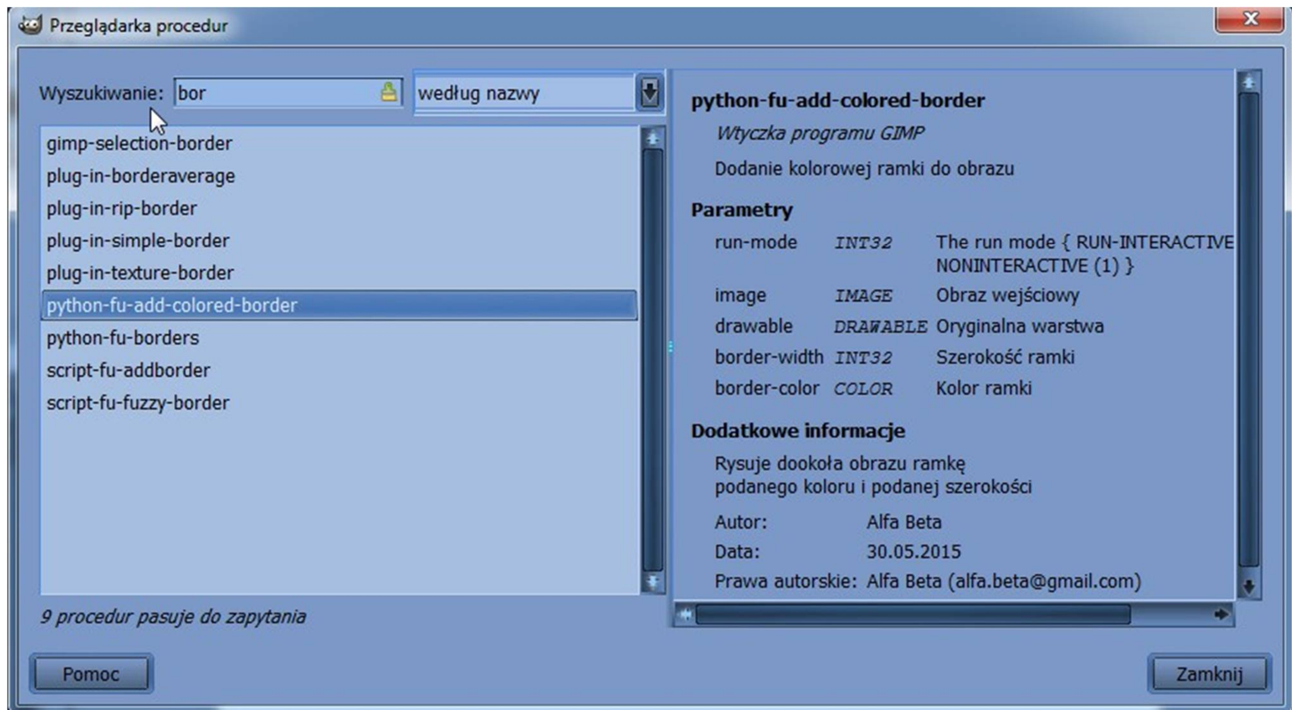
Możemy przejść do PDB i poszukać "**border**", aby znaleźć funkcję:

"python-fu-add-colored-border",

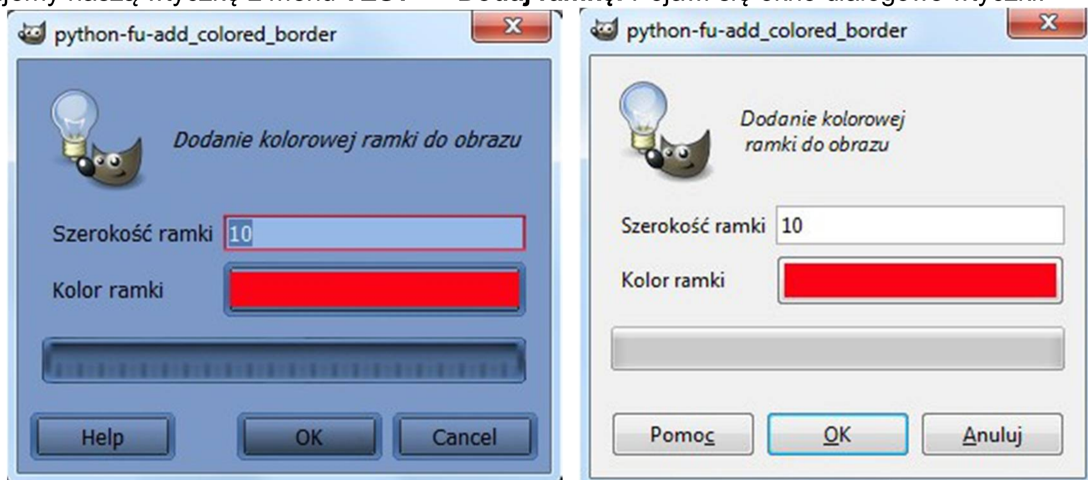
która pokazuje informacje jakie zapisaliśmy w naszej wtyczce.



W otwartym oknie PDB jak widać pojawiła się nowa procedura "**python-fu-add-colored-border**".

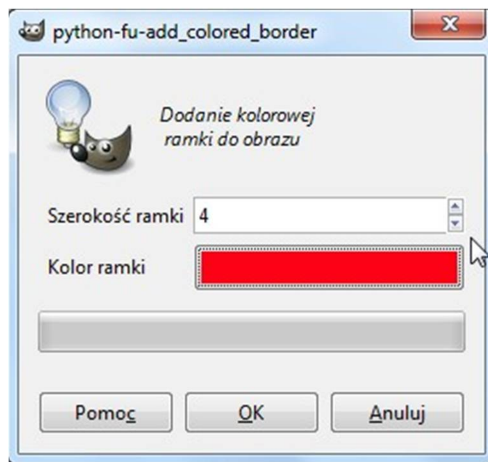


Wypróbujemy wtyczkę. W tym celu **Plik => Nowy…** ustalamy w otwartym oknie parametry nowego obrazu. Wywołujemy naszą wtyczkę z menu **TEST => Dodaj ramkę.** Pojawi się okno dialogowe wtyczki:

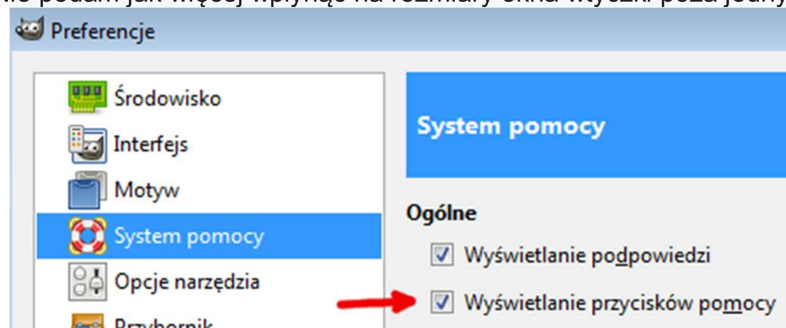


Widżet Kolor ramki – działa GIMP 2.8.6 32bit i 64bitPartha, 2.8.14 32bit; 2.8.10 64 bit aljacom, w wersjach 2.8.14 64bit standalone; 2.8.15 64bit Portable crash.

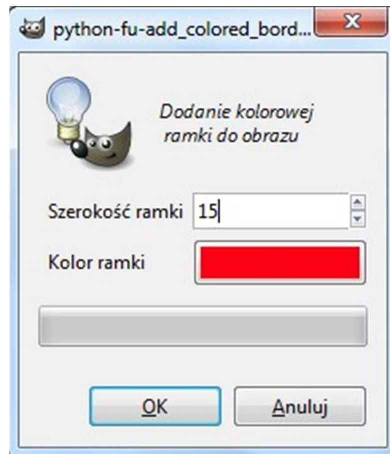
Jak widać GIMP sam postarał się o graficzny interfejs okna, i nie musimy wykonać brudnej roboty, rozmieszczając pola tekstowe i przyciski w oknie dialogowym, ale nie jesteśmy w stanie wpłynąć na zewnętrzne wymiary tego okna, [zmienłem nawet ("Dodanie kolorowej**\n** ramki do obrazu ")].



Zamiana `(PF_INT, "border_width", "Szerokość ramki", "10"),`
Na `(PF_SPINNER, "border_width", "Szerokość ramki", 10, (1, 100, 1)),`
Nie podam jak więcej wpłynąć na rozmiary okna wtyczki poza jednym:



Możemy w swoim GIMP odznaczyć, bo generalnie jest to większości niepotrzebne i otrzymamy okno:



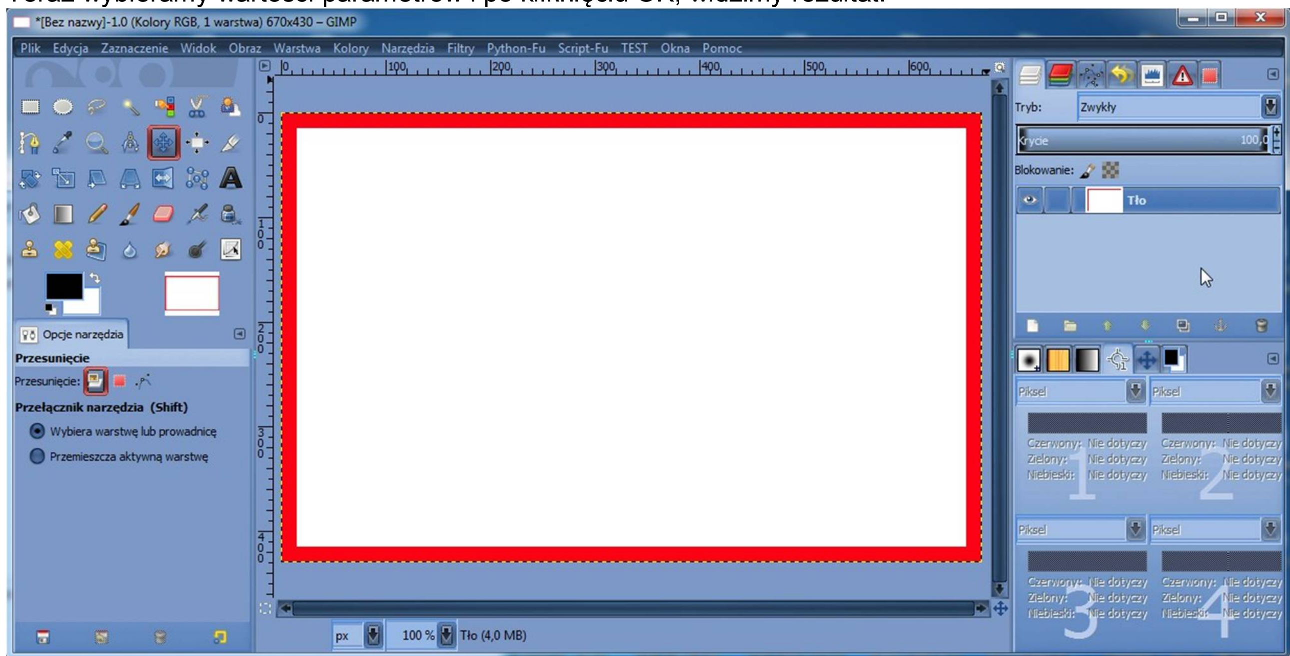
(Dla chcących poznać głębiej zagadnienie pyGTK layouts podaję przykładowy link:
http://www.yolinux.com/TUTORIALS/PyGTK.html

Co to jest GTK?

Ogólnie rzecz biorąc GTK jest to biblioteka do tworzenia graficznego interfejsu użytkownika (GUI). Biblioteka jest rozpowszechniania na licencji GPL/LGPL. Używając tej biblioteki można tworzyć programy darmowe, komercyjne, open-source, czy jakie tam kto sobie wymyśli ;).
Nazwa biblioteki pochodzi **G**IMP **t**ool**k**it, bo początkowo została stworzona dla GIMP'a.
Jej autorami są: Peter Mattis Spencer Kimball Josh MacDonald. GTK ma obiektowo zorientowany interfejs.
GtkWidget to klasa do tworzenia okienek, przycisków, pól edycyjnych, napisów itd.)

[Trwają prace nad zmianami:
https://developer.gnome.org/libgimp/stable/
https://developer.gnome.org/libgimp/stable/libgimpui.html a tutaj
https://developer.gnome.org/libgimp/stable/gimpcolorbutton.html]

Teraz wybieramy wartości parametrów i po kliknięciu OK, widzimy rezultat.



`Poprawność tłumaczenia border:` **granica = brzeg, obramowanie oraz ramka, krawędź**

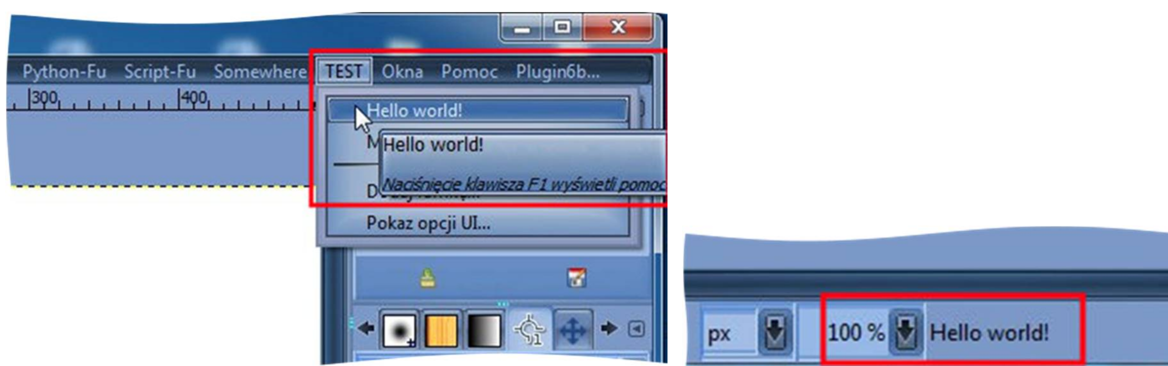
Dalej
prezentacja gotowych przykładów wraz z omówieniem najważniejszych fragmentów kodu.

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.

Zaczniemy od prostej wtyczki wyświetlającej komunikat, który pojawia się po umieszczeniu wskaźnika myszki na Etykiecie w menu, oraz w pasku stanu w dole okna Edytora obrazów:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# ------------------------------------------------------------------------------
-------
# Ten plik jest przykładem podstawowej wtyczki Python dla GIMP.
# Może być wykonana poprzez wybranie z menu opcji: "/TEST/Hello world!" (Witaj,
świecie!)"

from gimpfu import *

def Display_hello_world(img, layer) :
    '''
      Wtyczka nic nie robi, tylko po ustawieniu wskaźnika myszki na
      Etykiecie (nazwie punktu w menu), obok wskaźnika oraz na pasku
      stanu w dole okna Edytora obrazów GIMP wyświetli się komunikat
      "Hello World!"

    Parametery:
    img : image - Aktualny obraz.
    layer : warstwa aktualnego obrazu.
    '''
    gimp_message("Hello world!")  # Wyświetla komunikat w oknie dialogowym
                                                    # Przydatne do statusu lub
raportowania błędów.
                                                    # Komunikat musi być w UTF-8
```

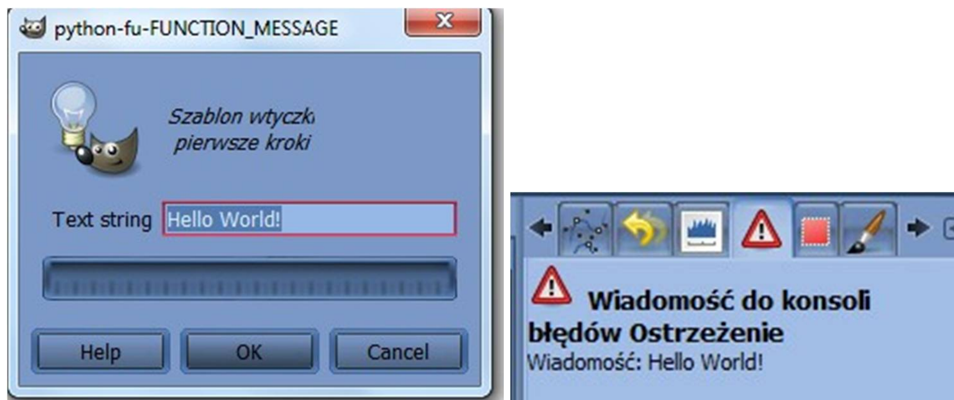
```
register(
    "test_hello_world",
    "Hello world!",
    "Wyświetla komunikat 'Hello world!' ",
    "Zbyma",
    "Open source",
    "2015",
    "Hello world!", # Etykieta, nazwa punktu w menu, za pomocą której, wtyczka
będzie uruchamiana,

    "*", # Typ obrazu: z którym wtyczka może współpracować, przy tym oznaczeniu
Nasza
             # wtyczka obsługuje wszystkie typy obrazów (ale przy tym
oznaczeniu,
             # bez obrazu Etykieta będzie wyszarzona), jeśli mamy puste "",
             # wtyczka jest dostępna nawet wtedy, gdy nie ma otwartego obrazu.
    [],
    [],
    Display_hello_world, menu="<Image>/TEST/" # Nazwa podana w def wywoływana
      # Naszym kodem oraz Ścieżka do punktu menu GIMP-a, gdzie powinna się
znajdować
      # Etykieta wtyczki, (ścieżka do menu musi startować z np. <Image> lub
<Toolbox>
      # czy <Layers>, wszędzie gdzie czujemy jej miejsce)
      )
main()
```



Kolejna wtyczka pokazująca przekazywanie wiadomości do Konsoli błędów:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from gimpfu import *

def plugin_message(image, layer, message):
    gimp.message("Wiadomość: " + message)  # wiadomość pojawi się
      # w otwartej konsoli błędów po kliknięciu OK, tekst tej wiadomości
      # można zmieniać

register(
        "FUNCTION_MESSAGE",
        "Szablon wtyczki\n pierwsze kroki",
        "Wskazane trochę bardziej szczegółowe wyjaśnienie",
        "Zbyma72age",
        "copyright",
        "2015",
        "<Image>/TEST/Wiadomość do konsoli błędów...", # Ścieżka i Etykieta
pozycji w menu
        "*", # typy obrazu: "RGB*, GRAY*"
        [ # argumenty funkcji (type, name, description, default [, extra])
```

```
            (PF_STRING, "string", "Text string", 'Hello World!')
        ],
        [], # wartość zwracana
        plugin_message) # nazwa funkcji

main()
```



Ponieważ GIMP w systemie Windows nie bardzo wie co zrobić z zastosowaniem funkcji drukowania "print" dlatego pokazano jak Definiujemy funkcję wyjścia "gprint", aby przekierowała wiadomości do Konsoli błędów.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Importujemy niezbędne moduły
from gimpfu import *

# GIMP w systemie Windows nie bardzo wie co zrobić z zastosowaniem
# funkcji drukowania "print" dlatego: Definiujemy funkcję wyjścia "gprint",
# aby przekierowała wiadomości do Konsoli błędów.
def gprint( text ):
    pdb.gimp_message(text)
    return

# Definiujemy Naszą funkcję, która będzie wykonywać faktyczne czynności
def plugin_function(image, drawable, text_value, int_value) :

    gprint("Hello World! ")
    gprint("Wyślij ten tekst: "+text_value)
    gprint("Wyślij tą cyfrę: %d"%int_value)
    gprint("Umożliwia również odwrócenie obrazu!")
    pdb.gimp_image_flip( image, ORIENTATION_HORIZONTAL )

    return

# Rejestrujemy wtyczkę w PDB
register(
    "plugin_debug", # Nazwa rejestrowanej wtyczki w Przeglądarce procedur,
      # oraz nazwa pojawiająca się na krawędzi okna wtyczki
    "Pierwsza wtyczka Python-Fu ",
    "Ta wtyczka nic nie robi, poza wysłaniem komunikatów do Konsoli błędów",
    "Michel Ardan, Modified Zbyma72age",
    "Michel Ardan (alfa.beta@gmail.com) Company GPL",
    "April 2010, Modyfied 2015",
    "Przekierowanie komunikatów drukowania do konsoli błędów...", # Etykieta
    "",
    [
      (PF_IMAGE, "image", "Input image", None),
      (PF_DRAWABLE, "drawable", "Input drawable", None),
        (PF_STRING, 'some_text', 'Wprowadź jakiś tekst', 'Piszemy cokolwiek'),
      (PF_INT, 'some_integer', 'Wprowadź jakąś cyfrę', 2010)
    ],
```

```
    [],
    plugin_function, menu="<Image>/TEST/Testowanie/"
    )
"""
 Otwieramy Konsolę błędów, uruchamiamy tą wtyczkę i patrzymy
 na to, co otrzymamy jako wyjście do Konsoli błędów. Odwrócenie (odbicie)
 obrazu widać natychmiast.
 """
main()
```

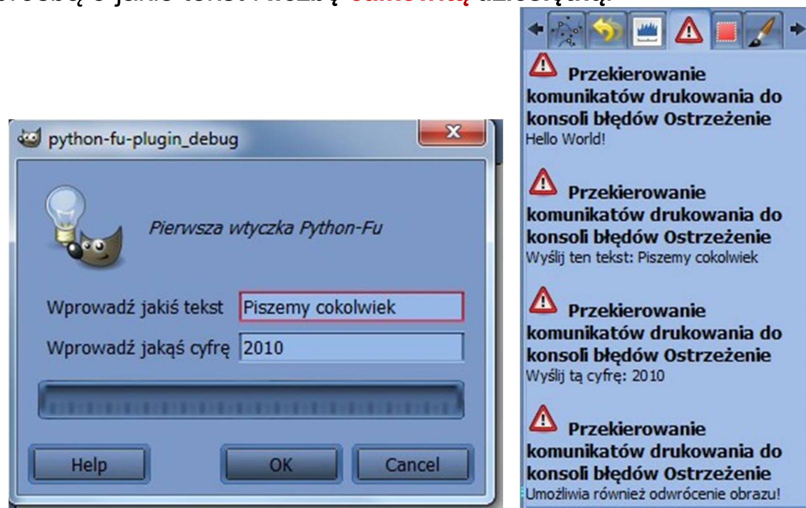
Przypomnienie:
Dla wtyczek, wykonujących operacje na obrazie, które można wywołać tylko wtedy, gdy użytkownik otworzy plik obrazu, czyli - **podległych obrazowi**, w `python-fu-register` należy podać dwa najważniejsze argumenty zmiennych typu:

`PF-IMAGE` i **`PF-DRAWABLE`**.

Dla **`PF-IMAGE`** i **`PF-DRAWABLE`**, powinna być przez nas podana wartość 0 lub None;
GIMP wypełni je ID bieżącego obrazu lub warstwy.
Wtyczka jest wywoływana w "run mode":

- interactive: pokazuje **GUI**, okno dialogowe ustawień dla użytkownika, aby zmodyfikować parametry
- non-interactive: działa z przekazanymi parametrami i nie powinna wchodzić w interakcje z użytkownikiem

Możemy użyć **Konsoli błędów** GIMP-a, znajdującej się w **"Okno => Dokowalne okna dialogowe => Konsola błędów"**.
Uruchamiamy GIMP-a i następnie po otwarciu obrazu uruchamiamy wtyczkę, zobaczymy interfejs Naszej wtyczki z prośbą o jakiś tekst i **liczbę całkowitą dziesiętną**.



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

http://gimpbook.com/scripting/gimp-script-templates/helloworld.py :
http://www.efalk.org/Docs/Python/gimp-examples.html

Plik Oryginalny:

```
#!/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)
```

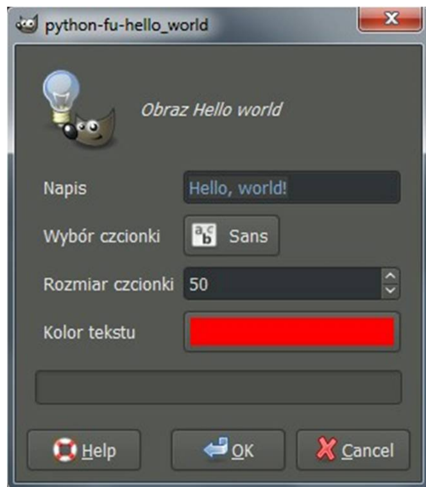
```
    # Save the current foreground color:
    pdb.gimp_context_push()

    # 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()
```

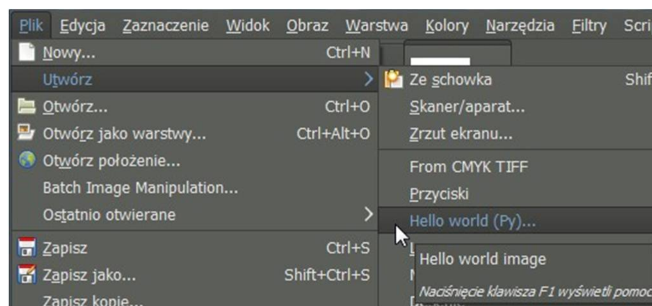
Oraz plik powyższej wtyczki **Hello World** *Akkana Peck*, z dodanymi komentarzami, a także poddany operacji ***lokalizacji***.
Plik wtyczki możemy skopiować i otworzyć w **Notepad++**:

```
#!/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?"
```

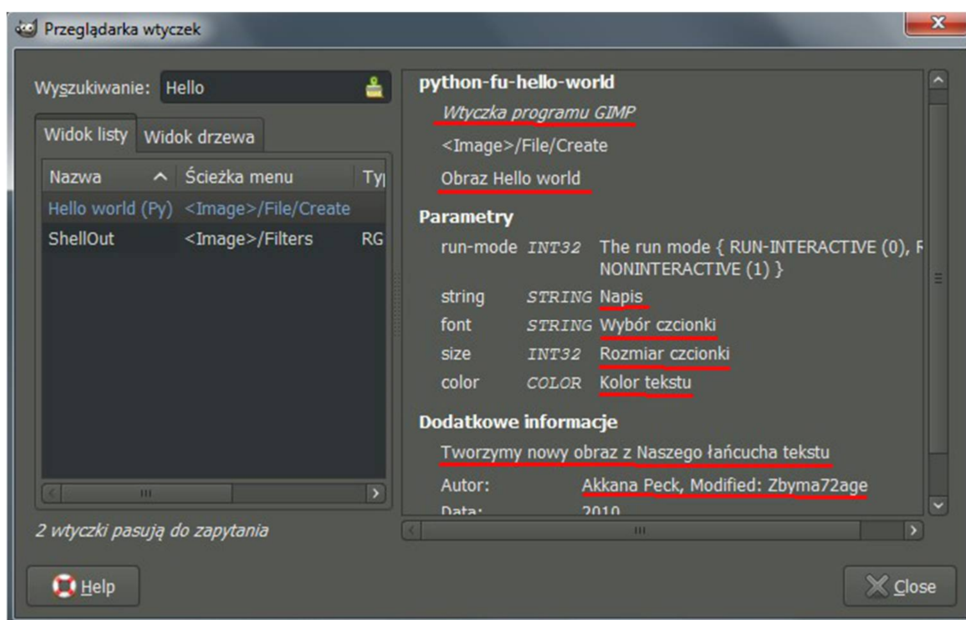
```
    # Tworzymy nowy obraz. Tymczasowo o rozmiarze 10x10 -- rozmiar będzie późnej
zmieniany.
    img = gimp.Image(10, 10, RGB)

    # 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()
```





Hello, world!



Powyżej pokazano jak w *Przeglądarce wtyczek* wyglądają te przetłumaczone istotne *stringi* – ciągi.

Kolejny przykład także znanej wtyczki James`a Henstridge **sphere.py**:
Źródło: https://raw.githubusercontent.com/goblin/gimp/master/plug-ins/pygimp/plug-ins/sphere.py lub
http://linuxsociety.org/pub/slack-stuff/gimp-psb/gimp-2.8.2/plug-ins/pygimp/plug-ins/sphere.py

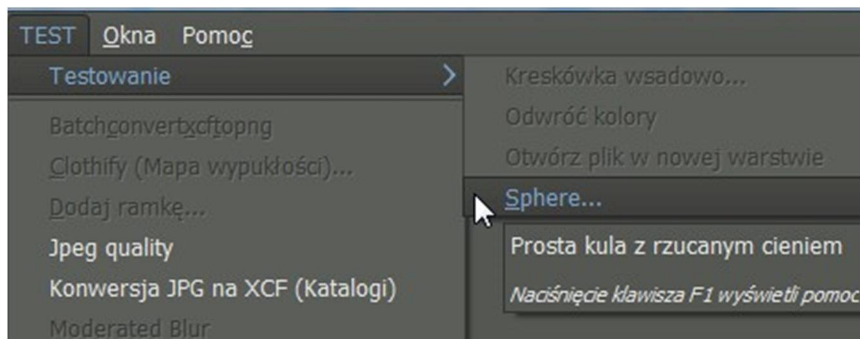
Twórca najbardziej kompletnego opracowania na temat **Dokumentacji GIMP Python**
http://www.gimp.org/docs/python/index.html
pracuje dla Canonical Ltd na Launchpad.
Jednym z jego hobby jest programowanie. Jest zaangażowany w wiele
Projektów programów z free software w tym Gnome (spec gnome-python).

James Henstridge
Oxford, 15 August 2004

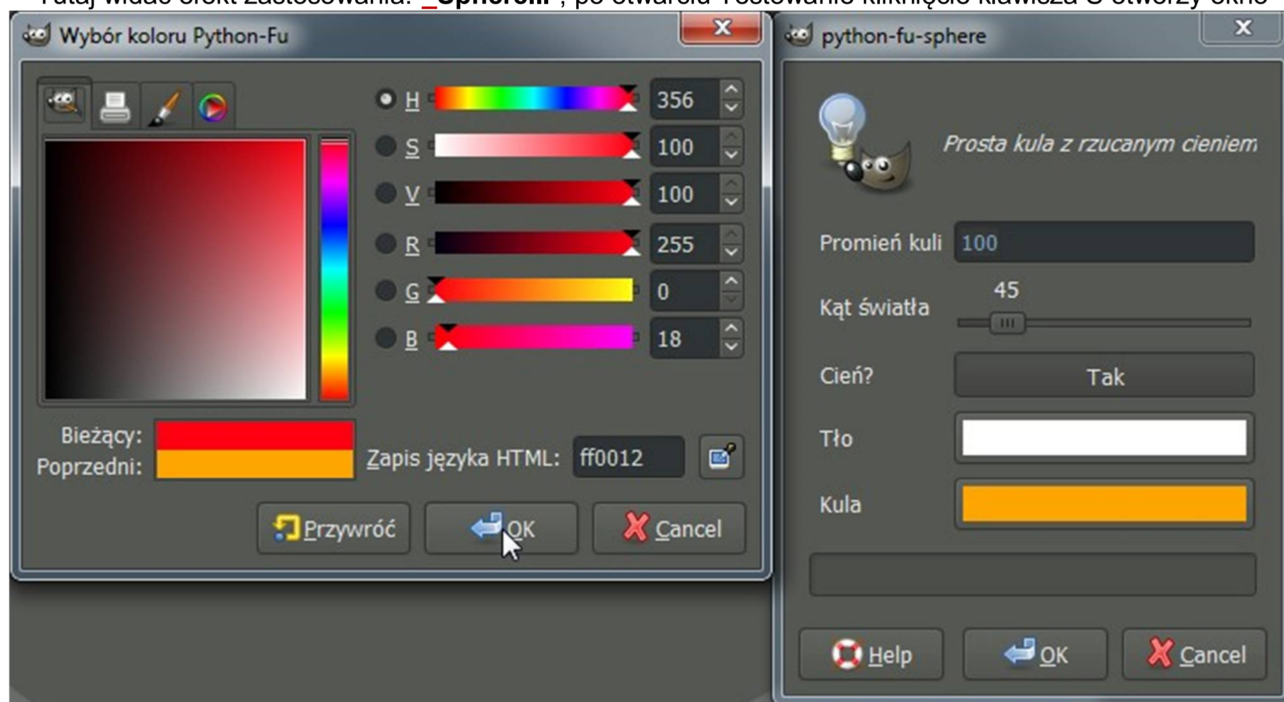
```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#   Gimp-Python - allows the writing of Gimp plugins in Python.
#   Copyright (C) 1997  James Henstridge <james@daa.com.au>
#
#   This program is free software: you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 3 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, see <http://www.gnu.org/licenses/>.
#   Źródło: https://raw.githubusercontent.com/goblin/gimp/master/plug-
ins/pygimp/plug-ins/sphere.py
#
import math   # moduł math zawiera deklaracje wielu przydatnych funkcji
matematycznych
              #wtyczka importuje biblioteki math Pythona, ponieważ stosuje kilka
podstaw
              # trygonometrii i wykorzystuje wartość pi.
from gimpfu import *  # mówi Pythonowi skąd ma załadować moduły GIMP

def sphere(radius, light, shadow, bg_colour, sphere_colour):
    if radius < 1:
        radius = 1
    # Oblicza rozmiar płótna. int(wartość) zamienia liczby zmiennoprzecinkowe
      # na liczby całkowite
    width = int(radius * 3.75)
    height = int(radius * 2.5)

    gimp.context_push() # Zapisujemy aktualny kolor pierwszoplanowy, aby móc z
niego skorzystać potem:
```

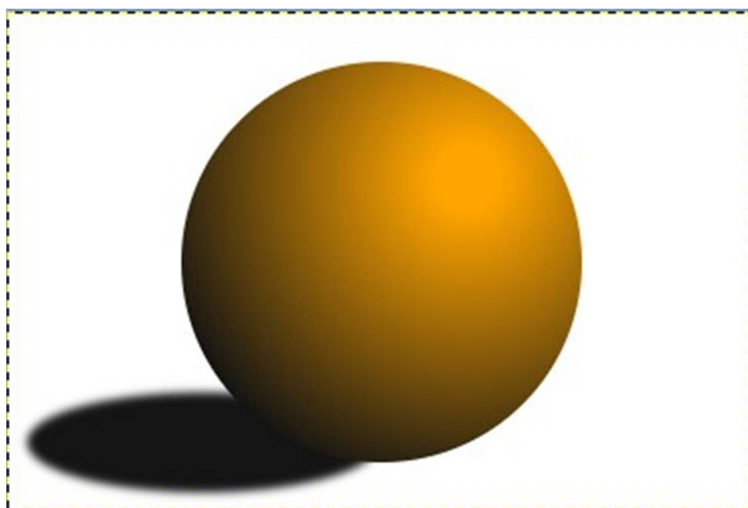
```
    img = gimp.Image(width, height, RGB)

    drawable = gimp.Layer(img, "Sphere Layer", width, height,
                          RGB_IMAGE, 100, NORMAL_MODE)  # Layer jest obiektem
tworzącym nową warstwę

    radians = light * math.pi / 180

    cx = width / 2
    cy = height / 2

    light_x = cx + radius * 0.6 * math.cos(radians)
    light_y = cy - radius * 0.6 * math.sin(radians)

    light_end_x = cx + radius * math.cos(math.pi + radians)
    light_end_y = cy - radius * math.sin(math.pi + radians)

    offset = radius * 0.1

    img.disable_undo()
    img.insert_layer(drawable) # procedura poprawiona - warstwa dodawana do
obrazu…

    gimp.set_foreground(sphere_colour) # Ustawiamy kolor na ten, który
przekazujemy do funkcji

    gimp.set_background(bg_colour)
    pdb.gimp_edit_fill(drawable, BACKGROUND_FILL)  # ...i wypełnia kolorem

    gimp.set_background(20, 20, 20)
    # stosowane poniżej operatory i ich znaczenie były opisane
    if (light >= 45 and light <= 75 or light <= 135 and
        light >= 105) and shadow:
        shadow_w = radius * 2.5 * math.cos(math.pi + radians)
        shadow_h = radius * 0.5
        shadow_x = cx
        shadow_y = cy + radius * 0.65

        if shadow_w < 0:
            shadow_x = cx + shadow_w
            shadow_w = -shadow_w

        pdb.gimp_ellipse_select(img, shadow_x, shadow_y, shadow_w, shadow_h,
                                CHANNEL_OP_REPLACE, True, True, 7.5)
        pdb.gimp_edit_bucket_fill(drawable, BG_BUCKET_FILL,
                                  MULTIPLY_MODE, 100, 0, False, 0, 0)

    pdb.gimp_ellipse_select(img, cx - radius, cy - radius, 2 * radius,
                            2 * radius, CHANNEL_OP_REPLACE, True, False, 0)
    pdb.gimp_edit_blend(drawable, FG_BG_RGB_MODE, NORMAL_MODE, GRADIENT_RADIAL,
                        100, offset, REPEAT_NONE, False, False, 0, 0, True,
                        light_x, light_y, light_end_x, light_end_y)

    pdb.gimp_selection_none(img)
    # procedura umożliwia cofnięcie stosu obrazu
    img.enable_undo()

      # 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
    disp = gimp.Display(img)
```

```
    gimp.context_pop()  # Przywracamy "stary" kolor pierwszego planu:


register(
    "sphere",
    "Prosta kula z rzucanym cieniem",
    "Prosta kula z rzucanym cieniem",
    "James Henstridge, Modified Zbyma72age",
    "James Henstridge",
    "1997-1999, Modified 2015",
    "_Sphere...",
    "",
    [
        (PF_INT, "radius", "Promień kuli", 100),
        (PF_SLIDER, "light", "Kąt światła", 45, (0,360,1)),
        (PF_TOGGLE, "shadow", "Cień?", 1),
        (PF_COLOR, "bg-color", "Tło", (1.0, 1.0, 1.0)),
        (PF_COLOR, "sphere-color", "Kula", "orange")
    ],
    [],
    sphere,  # Nazwa podana w def wywoływana Naszym kodem
    menu="<Image>/TEST/Testowanie" ) # ścieżkę do menu, można zmienić, wg
własnego uznania

main()
```



Tutaj widać efekt zastosowania:"**_Sphere...**", po otwarciu Testowanie kliknięcie klawisza S otworzy okno





Inna wtyczka tego samego autora tworzy wrażenie obrazu na płótnie:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#   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.

#   Wg. źródła http://www.gimp.org/docs/python/
#   Wtyczka jako Bump Map (nie w Python) dostarczana wraz z GIMP jest w menu:
#   Filtery → Artystyczne → Przekształcenie w tkaninę

from gimpfu import *  # mówi Pythonowi skąd ma załadować moduły GIMP

def clothify(timg, tdrawable, bx=9, by=9, azimuth=135, elevation=45, depth=3):
#  procedurze przekazywane sa wartości, uzyskane z okna dialogowego podane w
register;
#  timg i tdrawable informują o tym, że będą wykorzystywane otwarty obraz i
obszar roboczy
    width = tdrawable.width  # określamy szerokość i wysokość
    height = tdrawable.height

    img = gimp.Image(width, height, RGB)  # tworzymy obraz
    img.disable_undo()  # na czas pracy programu wyłączamy możliwość redagowania
        # obrazu przez użytkownika

    layer_one = gimp.Layer(img, "X Dots", width, height, RGB_IMAGE,
                           100, NORMAL_MODE)  # tworzymy warstwę, jej nazwa
będzie "X Dots"
    img.insert_layer(layer_one) # procedura poprawiona - warstwa dodawana do
obrazu…
    pdb.gimp_edit_fill(layer_one, BACKGROUND_FILL)  # ...i wypełnia kolorem
```

```
    pdb.plug_in_noisify(img, layer_one, 0, 0.7, 0.7, 0.7, 0.7)  # wykorzystujemy
wtyczkę
        # o odpowiedniej nazwie i przekazujemy do niej parametry

    layer_two = layer_one.copy()  # tworzymy jeszcze jedną warstwę - kopiowaniem
    layer_two.mode = MULTIPLY_MODE  # ustawiamy tryb warstwy
    layer_two.name = "Y Dots"  # ustalamy nazwę warstwy na "Y Dots"
    img.insert_layer(layer_two) # procedura poprawiona - warstwa dodawana do
obrazu…

    pdb.plug_in_gauss_rle(img, layer_one, bx, 1, 0)  # wykorzystujemy wtyczkę
Rozmycia Gauss
    pdb.plug_in_gauss_rle(img, layer_two, by, 0, 1)

    img.flatten()  # spłaszczamy wszystkie warstwy

    bump_layer = img.active_layer  # tworzymy zmienną z aktywnej warstwy

    pdb.plug_in_c_astretch(img, bump_layer)  # rozciągamy kontrast dla pokrycia
max możliwego zakresu
    pdb.plug_in_noisify(img, bump_layer, 0, 0.2, 0.2, 0.2, 0.2)
    pdb.plug_in_bump_map(img, tdrawable, bump_layer, azimuth,
                         elevation, depth, 0, 0, 0, 0, True, False, 0)

    gimp.delete(img)

register(
        "clothify", # Nazwa rejestrowanej wtyczki w Przeglądarce procedur, jeśli
tak wpiszemy, oraz nazwa na krawędzi okna GUI
               # GIMP doda prefiks "python-fu (lepiej prefiksu samemu nie
dodawać).
        "Tworzy wygląd warstwy\n podobny do drukowania\n na płótnie ",  # Krótki
opis działań wykonywanych przez
               # wtyczkę, zobaczymy go po umieszczeniu wskaźnika myszki na
nazwie punktu w menu.
        "Tworzy wygląd warstwy podobny do drukowania na płótnie ",  # Nazwa
Dokumentacyjna,
               # informacja o wtyczce, która pojawia się w Przeglądarce
procedur, Dodatkowe informacje -
               # wskazane wyjaśnić w bardziej szczegółowy sposób, co wtyczka
robi.
        "James Henstridge, Modified Zbyma72age", # Informacja o Autorze wtyczki
        "James Henstridge", # Informacja o prawach Autorskich
        "1997-1999, Modified 2015",  # data utworzenia wtyczki
        "_Clothify (Mapa wypukłości)...",  # Etykieta
        "RGB*, GRAY*",  # Typ obrazu: z którym wtyczka może współpracować, przy
tym oznaczeniu Nasza wtyczka obsługuje wszystkie
                # typy obrazów (ale przy tym oznaczeniu, bez obrazu Etykieta
będzie wyszarzona), alternatywnie można "*".
        [
            (PF_IMAGE, "image", "Input image", None),
            (PF_DRAWABLE, "drawable", "Input drawable", None),
            (PF_INT, "x-blur", "Rozmycie X", 9),  # parametry początkowe w oknie
wtyczki
            (PF_INT, "y-blur", "Rozmycie Y", 9),
            (PF_INT, "azimuth", "Azymut", 135),
            (PF_INT, "elevation", "Wzniesienie", 45),
            (PF_INT, "depth", "Głębia", 3)
        ],
        [],
        clothify, menu="<Image>/TEST")  # Nazwa podana w def wywoływana
        # Naszym kodem oraz Ścieżka do punktu menu GIMP-a, gdzie powinna się
znajdować Etykieta wtyczki,
```

```
        # używamy wielokropka ... w etykiecie otwierającej okno dialogowe
wtyczki,
        # (ścieżka do menu może startować z np. <Image> lub <Toolbox> czy
<Layers>, wszędzie
        # gdzie czujemy jej miejsce)

main()
```

Zakres wartości:
3 do 100
3 do 100
(PF_INT, "azimuth", "Azymut", 135), 0 do 360
(PF_INT, "elevation", "Wzniesienie", 45), 0 do 90
(PF_INT, "depth", "Głębia ", 3) 1 do 50

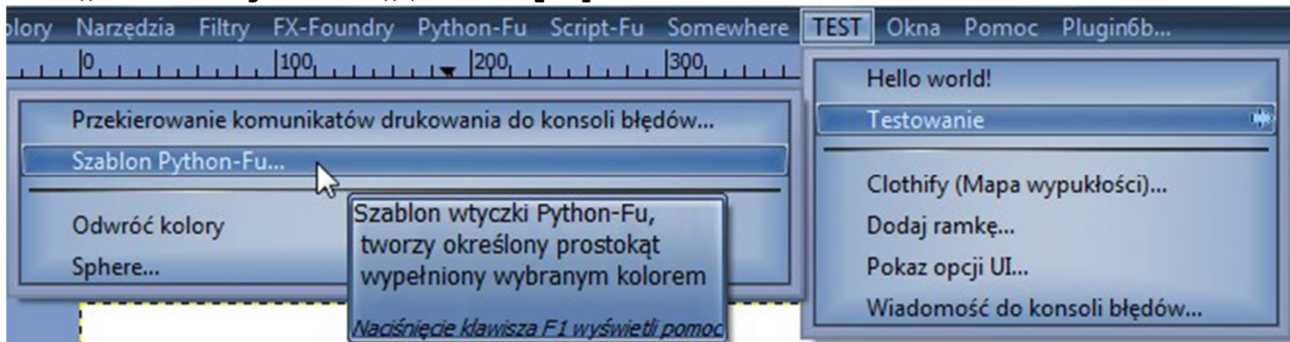
Na zakończenie jeszcze trzy przykłady:

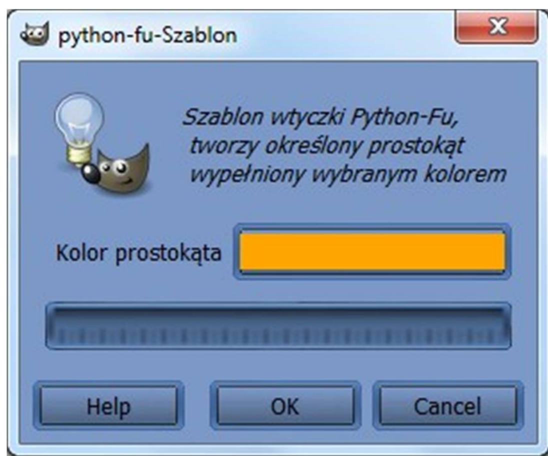
1. Szablon wtyczki, tworzącej kolorowy prostokąt na istniejącym obrazie

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
#
# Program jest wolnym oprogramowaniem: możesz go rozprowadzać dalej i / lub
modyfikować
# je na warunkach licencji GNU General Public License, wydanej przez
# Free Software Foundation; według wersji 3 tej Licencji lub
# (Do wyboru) którejś z późniejszych wersji.
#
# Program jest rozpowszechniany w nadziei, że będzie użyteczny,
#, Ale BEZ JAKIEJKOLWIEK GWARANCJI; nawet bez gwarancji
# Handlowej lub przydatności do KONKRETNEGO CELU. Zobacz
# Licencja Publiczna GNU więcej szczegółów.
#
# GNU General Public License, patrz <http://www.gnu.org/licenses/>.

from gimpfu import *  # Importujemy niezbędne moduły (odpowiednie biblioteki),
          # wtyczka musi wiedział, z których wbudowanych funkcji ma korzystać.

def python_fu_template(image, drawable, color):

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

    image.undo_group_start()  # funkcja służy do rozpoczęcia cofania grupy,
                                # stosowany w połączeniu z nazwą
'image.undo_group_end()'.
                                            # Zakazujemy zapisu informacji dla
umżliwienia cofnięć,
  # aby wszystkie operacje wykonane wtyczką można było anulować jednym klikiem
  # klikając Ctrl + Z lub wybierając w menu "Edycja" punkt "Cofnij"

    (x0, y0) = drawable.offsets  # przesunięcie, wyrównanie drawable,
      # Użyjemy zamiast nieaktualnego: 'gimp-image-select-rectangle'.
    pdb.gimp_rect_select(image,
                         x0 + drawable.width  / 4,  # x coordinate of upper-left
corner of rectangle
                         y0 + drawable.height / 4,  # y coordinate of upper-left
corner of rectangle
                         drawable.width  / 2,       # The width of the rectangle
(width >= 0)
                         drawable.height  /  2,         # The height of the
rectangle (height >= 0)
                         CHANNEL_OP_REPLACE, 0, 0)   # Operation The selection
operation
                                                                   # Feather option
for selections (TRUE or FALSE)
                                                                   # Radius for
feather operation (feather-radius >=

    gimp.set_foreground(color)  #  Ustawiamy kolor na ten, który przekazujemy do
funkcji
    pdb.gimp_edit_fill(drawable, FOREGROUND_FILL)    # zaznaczenie prostokąta
wypełniamy kolorem

    image.undo_group_end()  # funkcja służy do zakończenia cofania grupy

    gimp.displays_flush()  # Aktualizujemy, odświeżamy obraz na wyświetlaczu

    gimp.context_pop()  # Przywracamy "stary" kolor pierwszego planu

register(
    "Szablon",  # Nazwa rejestrowanej wtyczki w Przeglądarce procedur, jeśli tak
wpiszemy, oraz nazwa na krawędzi okna GUI
            # GIMP doda prefiks "python-fu (lepiej prefiksu samemu nie dodawać).
```

```
    "Szablon wtyczki Python-Fu,\n tworzy określony prostokąt\n wypełniony
wybranym kolorem ", # Opis okna wtyczki,
    "Szablon wtyczki, tworzy kolorowy prostokąt na istniejącym obrazie",   #
Nazwa Dokumentacyjna, szerszy opis, pojawi się w PDB - Dodatkowe informacje.
    "Zbigniew Malach, Zbyma72age",  # Autor
    "Zbigniew Malach GPL",  # Informacja o prawach Autorskich
    "2015-05-30",  # data utworzenia wtyczki
    "Szablon Python-Fu...",  # Etykieta
    "RGB* GRAY*",  # Typ obrazu: z którym wtyczka może współpracować, przy tym
oznaczeniu Nasza wtyczka obsługuje wszystkie
             # typy obrazów (ale przy tym oznaczeniu, bez obrazu Etykieta będzie
wyszarzona), alternatywnie można "*".
    [
        (PF_IMAGE,    "image",     "Input Image",      0),  # możemy podać 0 lub
None
        (PF_DRAWABLE, "drawable", "Input Drawable",  0),
        (PF_COLOR,    "color",    "Kolor prostokąta", "orange")
    ],
    [],
    python_fu_template,
    menu="<Image>/TEST/Testowanie/")  # Nazwa podana w def wywoływana Naszym
kodem.
      # Ścieżka do punktu menu GIMP-a, gdzie powinna się znajdować Etykieta
wtyczki,
      # używamy wielokropka ... w etykiecie otwierającej okno dialogowe wtyczki,
      # (ścieżka do menu może startować z np. <Image> lub <Toolbox> czy
<Layers>, wszędzie
      # gdzie czujemy jej miejsce)

main()  # funkcji main (), może być pusta
```





2. Wtyczka, którą jednym kliknięciem Etykiety, odwracamy kolory otwartego obrazu.
Oczywiście, jest w GIMP zaimplementowana taka procedura przez GIMP Development Team
Nazwa **`color-rotate`** `"OriginalFilename", "color-rotate.exe"`

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# --------------------------------------------------------------------------------
-------
# Na podstawie: test-invert-layer.py
# Copyright (c) 2013, Jose F. Maldonado
# All rights reserved.
#
# Ten plik jest podstawowym przykładem wtyczki Python dla GIMP.
#
# To może być wykonane poprzez wybranie opcji w menu: '/TEST/Testowanie/Odwróć
kolory'


from gimpfu import *

def invert_drw(img, drawable) :
    ''' Odwraca kolory otwrtego obrazu.

    Parametry:
    img : image Aktualny obraz.
    drawable : drawable aktualnego obrazu.
    '''
    pdb.gimp_invert(drawable)

register(
    "test_invert_drw",
    "Odwraca kolory",
    "Jednym kliknięciem Etykiety, odwraca kolory otwartego obrazu",
    "JFM, Modified Zbyma72age",
    "Open source (BSD 3-clause license)",
    "2013, Modified 2015",
    "Odwróć kolory", # Etykieta
    "*",
    [],
    [],
    invert_drw,
      menu="<Image>/TEST/Testowanie"
      )
"""
Licencja typu BSD skupia się na prawach użytkownika. Jest bardzo liberalna,
zezwala nie tylko na modyfikacje kodu i jego rozprowadzanie w takiej postaci,
ale także na rozprowadzanie produktu bez postaci źródłowej czy wręcz włączenia
do zamkniętego oprogramowania, pod warunkiem załączenia do produktu informacji
o autorach oryginalnego kodu i treści licencji.
"""
main()
```

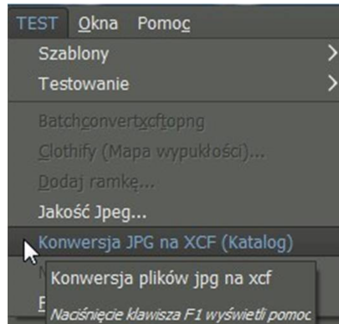
3. To jest przykład wtyczki, opublikowanej w formie Tutoriala na stronie:
http://www.gimp.org/tutorials/AutomatedJpgToXcf/
a także w formie: https://mail.gnome.org/archives/gimp-docs-list/2013-June/pdfle4afpT0Ft.pdf
Autorem jest: **Stephen Kiel** example-jpeg-to-xcf.py
Dodałem:
 # -*- coding: utf-8 -*-
Zmieniłem dla poradnika: **menu = "<Image>/TEST"**
Wprowadziłem lokalizację
Sprawdziłem działa poprawnie, ale tylko w gimp-2.8.6Portable32Partha i gimp-2.8.6Portable64Partha

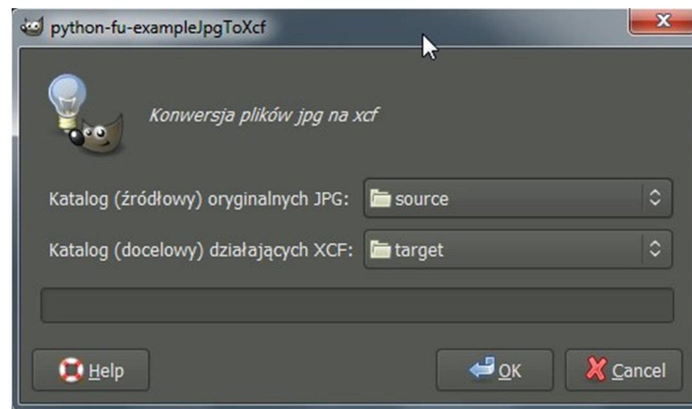
```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
#
# Plik pochodzi z:
# http://www.gimp.org/tutorials/AutomatedJpgToXcf/
#
# Sprawdzono pracuje poprawnie w:
# gimp-2.8.6Portable32Partha i 2.8.6Portable64Partha ,
#
# Ten program jest wolnym oprogramowaniem: możesz go rozprowadzać dalej
# i / lub modyfikować
# Zgodnie z warunkami licencji GNU General Public opublikowane przez
# Free Software Foundation; wersji 3 tej Licencji lub
# (Do wyboru) którejś z późniejszych wersji.
#
# Ten program jest rozpowszechniany w nadziei, że będzie użyteczny,
# Ale BEZ JAKIEJKOLWIEK GWARANCJI; nawet bez gwarancji
# Handlowej lub przydatności do KONKRETNEGO CELU. Zobacz
# Licencja Publiczna GNU więcej szczegółów.
#
# Powinieneś otrzymać kopię GNU General Public License
# Wraz z tym programem. Jeśli nie, patrz <http://www.gnu.org/licenses/>
#
# Uwagi Autora:
# Ta przykładowa wtyczka jest bardzo prosta; tworzenie listy nazw plików,
# prowadzenie przez te listy w pętli, otwarcie każdego pliku źródłowego i
# z kolei  jego eksportowanie. Korzystając z tego przykładu będziemy mieć
# możliwość jego wykorzystania jako modelu do budowy podobnych funkcji.
# Przykładem podobnej funkcji może być konwersja katalogu plików XCF do
# plików JPG, skalowanie oryginalnych plików jpg do mniejszego rozmiaru, itd.
#
# Podczas korzystania z niego GIMP przegląda system plików, prawdopodobnie
# trzeba będzie wybrać "Inne", aby poruszać się, tam gdzie naprawdę chcemy.
#
# W przypadku korzystania z automatycznej wtyczki zawsze dobrym pomysłem jest,
# aby uruchomić go na KOPII swoich oryginalnych obrazów.
# Wtyczka jest przykładem, który ma na celu zilustrowanie automatyzacji
# część procesu edycji.
#
# Więcej informacji w podanym powyżej linku do źródła.
#
# Uwaga: Nazwy katalogu zrodlowego i docelowego nie mogą zawierać
# polskich znaków diakrytycznych !!!!

###############################################################################
#
from gimpfu import *
import os
import re
#
def exampleJpgToXcf(srcPath, tgtPath):
    """Zarejestrowana funkcja exampleJpgToXcf,
      konwersja wszystkich plików JPEG w katalogu źródłowym
      na pliki XCF w katalogu docelowym.
```

```
      Wymaga dwóch argumentów, ścieżki do katalogów źródłowego i docelowego.
      Nie wymaga otwartego jakiegoś obrazu.
    """
    ###
    open_images, image_ids = pdb.gimp_image_list()
    if open_images > 0:
        pdb.gimp_message ("Zamknij otwarte obrazy i uruchom ponownie")
    else:
        # lista wszystkich plików w katalogach źródłowym i docelowym
        allFileList = os.listdir(srcPath)
        existingList = os.listdir(tgtPath)
        srcFileList = []
        tgtFileList = []
        xform = re.compile('\.jpg', re.IGNORECASE)
        # Znajdź wszystkie pliki z listy w formacie JPEG i utwórz nazwy plików
XCF
        for fname in allFileList:
            fnameLow = fname.lower()
            if fnameLow.count('.jpg') > 0:
                srcFileList.append(fname)
                tgtFileList.append(xform.sub('.xcf',fname))
        # Słownik - nazw plików źródłowych i docelowych
        tgtFileDict = dict(zip(srcFileList, tgtFileList))
        # Pętla na jpeg, otwórz każdy i zapisz jako xcf
        for srcFile in srcFileList:
            # Nie nadpisuj istniejących, może być praca w toku
            if tgtFileDict[srcFile] not in existingList:
                # os.path.join wstawia odpowiedni rodzaj separatora pliku
                tgtFile = os.path.join(tgtPath, tgtFileDict[srcFile])
                srcFile = os.path.join(srcPath, srcFile)
                theImage = pdb.file_jpeg_load(srcFile, srcFile)
                theDrawable = theImage.active_drawable
                pdb.gimp_xcf_save(0, theImage, theDrawable, tgtFile, tgtFile)
                pdb.gimp_image_delete(theImage)
#
############################################################################
#
register (
    "exampleJpgToXcf",             # Nazwa rejestrowanej wtyczki w Przeglądarce
procedur
    "Konwersja plików jpg na xcf ", # Opis okna wtyczki
    "Konwersja plików jpg na xcf ", # Nazwa Dokumentacyjna
    "Stephen Kiel Modified Zbyma72age",         # Author
    "Stephen Kiel",             #  Informacja o właścicielu prawach autorskich
(Copyright)
    "July 2013 Modified 07.2015",             # data utworzenia wtyczki
    "Konwersja JPG na XCF (Katalog)", # Etykieta
    "",        # Typ obrazu: z którym wtyczka może współpracować, nie wymaga
otwarcia obrazu
    [
    ( PF_DIRNAME, "srcPath", "Katalog (źródłowy) oryginalnych JPG:", "" ),
    ( PF_DIRNAME, "tgtPath", "Katalog (docelowy) działających XCF:", "" ),
    ],
    [],
    exampleJpgToXcf,   #  Nazwa podana w def, wywoływana kodem wtyczki
    menu="<Image>/TEST"  # Lokalizacja w menu
    )   # Koniec register

main()
```

Tworzę na pulpicie dwa katalogi, w surce umieszczam przykładowe plik jpg.

Pliki źródłowe





Na tym zakończę:

Wprowadzenie do podstaw programowania, czytania i spolszczania wtyczek Python-Fu.

Jest jeszcze dużo więcej zagadnień, które wymagają poznania, ale to nie jest podręcznik.
Myślę, że będzie to pomocny poradnik do zrozumienia niektórych określeń stosowanych we wtyczkach.
Gdyby pojawiły się wątpliwości, dlaczego przytaczam przykłady innych autorów, odpowiadam, bo są mądrzejsi niż ja.

Zbiór przykładowych wtyczek: http://zbyma.gimpuj.info/Szablony%20niektorych%20wtyczek.zip

Autor opracowania:
inż. Zbigniew Małach
Zbyma72age

Wszelkie prawa zastrzeżone.
Poradnik nie może być publikowany w całości lub fragmentach na innych stronach www lub prasie, bez
wcześniejszego kontaktu z autorem poradnika i pisemnej zgody na publikację