

Język Python na potrzeby GIS

Katedra Geoinformacji, Fotogrametrii i Teledetekcji Środowiska

©2019 Mariusz Twardowski

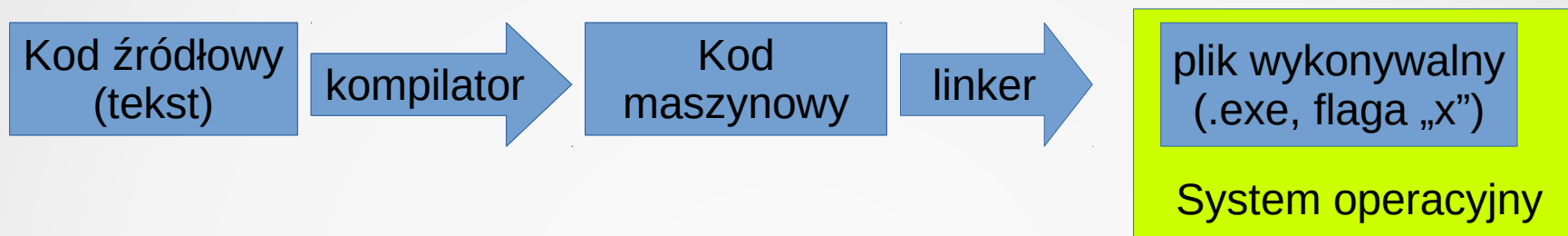
LibreOffice Impress
<http://libreoffice.org>

Wspomaganie komputerowe GIS

- **Programy** – samodzielne aplikacje działające w określonym środowisku,
- **Skrypty** – pliki tekstowe wymagające do uruchomienia odpowiedniego interpretera,
- **Biblioteki** – zbiory funkcji nie uruchamiane bezpośrednio, ale możliwe do wykorzystania w programach lub skryptach,
- **Wtyczki** – pliki uruchamiane poprzez specyficzną aplikację dla której zostały stworzone, mogą mieć postać bibliotek lub skryptów.

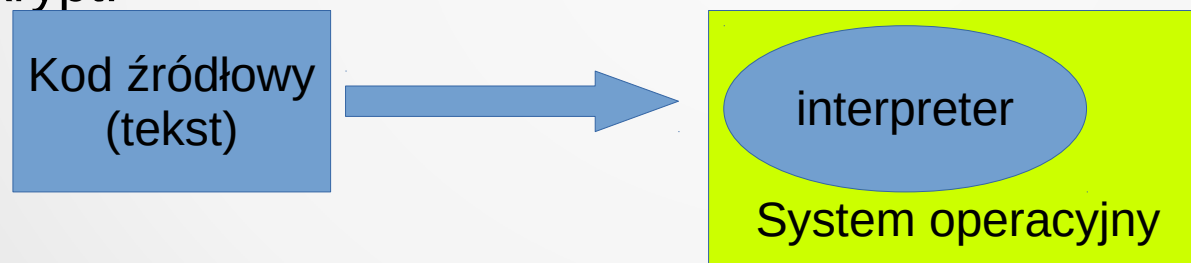
Program vs skrypt

- Program:



zalety – szybkość i spójność wykonania, pamięć

- Skrypt:



zalety – wieloplatformowość, łatwe testowanie kodu

Środowisko uruchomienia

- QGIS 2.x – <http://qgis.org>, zawiera:
 - implementację Python i zintegrowana konsolę
 - bibliotekę pyQT do interfejsu graficznego
 - środowisko OSGeo4W
- Notepad++ – <http://notepad-plus-plus.org>,
 - zaawansowany edytor tekstu ASCII
 - podświetlanie składni
 - wtyczki, automatyczne uruchamianie skryptów

Wtyczki Python w GIS

- QGIS – <http://qgis.org>
 - program na licencji GPL,
 - wieloplatformowy: Windows, Linux, MacOS, BSD, Android
 - szybki rozwój w ostatnich latach,
 - coraz większa popularność w produkcji,
 - użycie biblioteki pyQT.
- ArcGIS – <http://esri.com>
 - licencja komercyjna, Windows,
 - standard produkcyjny,
 - wtyczki python, podobnie jak w QGIS.

Cechy języka Python

- stosunkowo krótka historia,
- wysoki poziom,
- uniwersalne zastosowanie,
- łatwość zrozumienia kodu,
- konserwatywna ilość kodu,
- programowanie strukturalne i obiektowe,
- wieloplatformowość,
- łatwy do nauczenia,
- popularność.

Podstawowe operatory

operator	znak
przypisanie	=
matematyczne	+ - * / ** %
porównania	== != > < <= >=
logiczne	and or not
przynależności	in not in
wyświetlanie wartości	print (python 2.7)

Proste typy danych

- int – wartości całkowite

x = 3764 # przypisanie obiektu 3764 do x

- str – ciągi znaków

nazwisko = „kowalski”

- float – wartości zmiennoprzecinkowe

liczbapi = 3.1415

wynik = 2.0

- sprawdzenie typu zmiennej

type(zmienna)

Ciągi

- oznaczane cudzysłowami lub apostrofami
- niemutowalne, niemożliwe zmiany bezpośrednie
- mechanizm cięcia (*slicing*), pozwala na modyfikację

'Litwo ojczyzno moja'[6:14] # 'ojczyzno'

nazwisko[:2] # 'Ko'

(nazwisko+' Jan')[-9:] # 'alski Jan'

- ciągi jako obiekty

nazwisko.upper() # 'KOWALSKI'

nazwisko.find('lsk') # 4

Typy złożone

- `lista = ['wpis 0', 'wpis 1', 2, 'wpis 3', 2.0]`
`lista[2] = 5` # `lista = ['wpis 0', 'wpis 1', 5, 'wpis 3', 2.0]`
- `tupla = ('wpis 0', 'wpis 1', 2, 'wpis 3', 2.0)`
`tupla[2:4]` # `(2, 'wpis 3')`
- `zbior = set(('wpis 0', 'wpis 1', 2, 'wpis 3', 2.0))`
`print(zbior)` # `set([2, 'wpis 3', 'wpis 1', 'wpis 0'])`
- `słownik = {0: 'wpis 0', 1: 'wpis 1', 'drugi': 2, 'trzeci': 'wpis 3'}`
`print(słownik[1])` # `'wpis 1'`
`print(słownik['drugi'])` # `2`
`słownik['trzeci']=3` # `{0: 'wpis 0', 1: 'wpis 1', 'drugi': 2, 'trzeci': 3}`

Operatory logiczne

- sprawdzanie warunków

`10 == 10` (prawda), `10 != 10` (fałsz), `10 <= 5` (fałsz)

- tabela prawdy and, or, not

`1 and 1 = 1`, `1 and 0 = 0`, `0 and 1 = 0`, `0 and 0 = 0`

`1 or 1 = 1`, `1 or 0 = 1`, `0 or 1 = 1`, `0 or 0 = 0`

`1 xor 1 = 0`, `1 xor 0 = 1`, `0 xor 1 = 1`, `0 xor 0 = 0`

`not 1 = 0`, `not 0 = 1`

- łączenie warunków

`10 == 10 or 10 >= 3` # prawda, jeden warunek spełniony

`3 < 4 and 4 < 10` # prawda, oba warunki spełnione

`3 < 4 < 10` # uproszczenie zapisu

Formatowanie ciągów (tradycyjne)

- znacznik formatowania '%':
 - %d** lub **%i** – liczby całkowite,
 - %s** – ciągi,
 - %c** – znak lub liczba,
 - %f** lub **%F** – liczby zmiennoprzecinkowe,
 - %u** – liczby naturalne
 - %%** - znak %
 - %x** lub **%X** - hex

Formatowanie ciągów - przykłady

- `print(' str - %s, int - %i, float - %f ') % ('abc', 108, 3.14)`
#wynik: str - abc, int - 108, float – 3.140000
- `print('znak: %c, hex: %X, procent %%') % (69, 385454)`
#wynik: znak: E, hex: 5E1AE, procent %
- `sl = {'d0':255,'d1':'abc','d2':34.81}`
`print('p1: %(d1)s, p2: %(d2).3f, p0: %(d0)X') % sl`
#wynik: p1: abc, p2: 34.810, p0: FF

Formatowanie ciągów (nowy styl)

- Użycie metody **str.format()**

```
print('War1:{} War2:{} War3:{}'.format('aaa', 3.0, 2))
```

```
#wynik: War1:aaa War2:3.0 War3:2
```

```
v1='Las'; v2=7; v3=2.6
```

```
print('zm1: {}, zm2: {}, zm3 {}'.format(v2,v1,v3))
```

```
#wynik: zm1: 7, zm2: Las, zm3 2.6
```

- Wykorzystanie funkcji **locals()**

```
print('zm1: {v3}, zm2: {v1}, zm3 {v2}'.format(**locals()))
```

```
#wynik: zm1: 2.6, zm2: Las, zm3 7
```

Sekwencje ucieczki znak '\'

- `\\` Backslash (`\`)
- `\'` Single-quote (`'`)
- `\"` Double-quote (`"`)
- `\n` ASCII linefeed (LF)
- `\r` ASCII Carriage Return (CR)
- `\t` ASCII Horizontal Tab (TAB)
- `\uxxxx` Character with 16-bit hex value `xxxx` (Unicode only)
- `\Uxxxxxxxx` Character with 32-bit hex value `xxxxxxxx` (Unicode only)
- `\ooo` Character with octal value `ooo`
- `\xhh` Character with hex value `hh`

przykład: `print('235 dziesiętnie to:\n %X szesnastkowo' % 235)`

#wynik: 235 dziesiętnie to:

EB szesnastkowo

Kodowanie znaków

- Standardowy zestaw ASCII (7 bits) nie zawiera znaków regionalnych takich jak ą ć ń
- Różne kraje mają przypisane różne kodowania, dla Polski jest to **iso-8859-2**
- Była potrzeba ujednolicenia kodowania więc powstało UTF-8, które zawiera większość krajów zachodnich.
- Możemy oznaczyć kodowanie na początku skryptu:
-*- coding: utf-8 -*-
- Możemy oznaczyć linię markerem 'u':
ourstr = u' Zżółć gęślą jaźń '

Systemy liczbowe

- dziesiętny

cyfry 0, 1, 2, 3 ... 9 rzędy wielkości: 10^0 10^1 10^2 10^3

$$255 = 2 * 10^2 + 5 * 10^1 + 5 * 10^0 = 200 + 50 + 5$$

- binarny/dwójkowy

cyfry 0 i 1 – bit najmniejsza ilość informacji

rzędy wielkości: 2^0 , 2^1 , 2^2 , 2^3 , ..., $2^7=128$, $2^8=256$

liczymy: 0,1,10,11,100,101,110,111,1000....11111111 (bajt, 255)

- szesnastkowy

cyfry 0,1,2...9,A,B,C,D,E,F

rzędy wielkości: $16^0=1$ (0x01), $16^1=16$ (0x10), $16^2=256$ (0x100)

liczymy: 0,1,2,..., 9, A,..., F, 10, 11,..., 19, 1A, ..., 1F, 20, ..., FF (bajt)

Podstawy języka Python

- dokumentacja:

<https://docs.python.org/2/reference/>

- komentarz:

znak # dla pojedynczej linii

znaki ''' dla bloku kodu

- konsola Pythona – pojedyncze wyrażenia
- skrypt – połączenie wielu poleceń
- bloki programu kontrolowane wcięciami

Skrypty i kontrola przepływu

- plik tekstowy z rozszerzeniem „.py”
- zbiór poleceń które chcemy wykonać

```
a = 3
b = 4
print 'Wynik : ', a+b
```
- wykonanie skryptu interpreterem

```
python test.py
```
- bloki programu oznaczane są wcięciami
- dowolne nazwy zmiennych z wyjątkiem słów kluczowych

Wcięcia – kontrola przepływu

-*- coding: cp1250 -*-

class Klasa1(object):

def metoda1(self, arg1, arg2):

#Blok kodu klasy

print 'metoda1'

#blok kodu metoda1

print 'należy do Klasa1'

print 'te same wcięcia'

def metoda2(self, arg4):

print 'metoda2 też należy do Klasa1'

#blok kodu metoda1

print 'te same wcięcia w całej metodzie'

print 'choć inne niż w poprzedniej'

def funkcja3(arg3):

#definicja poza blokiem klasy

print 'to nie jest metoda'

#blok kodu funkcja3

print 'i nie należy do Klasa1'

print 'tutaj zaczyna się program'

#blok programu głównego

ob1=Klasa1()

ob1.metoda1(1,3)

ob1.metoda2('test')

funkcja3(2)

ob1.funkcja3(2) **#Błąd, funkcja nie należy do klasy i obiekt nie posiada tej metody**

Wyrażenia warunkowe

W zależności od wartości zmiennej możliwe są różne ścieżki wykonania programu, struktura **if ... elif ... else** :

```
print 'Czy chcesz kontynuowac (t/n)?'
```

```
k=raw_input()
```

```
if k == 't':
```

```
    print 'Kontynuuję program'
```

```
elif k == 'n':
```

```
    print 'Zatrzymuję program'
```

```
    exit()
```

```
else:
```

```
    print 'Zła opcja'
```

```
print 'Dalsza część programu'
```

Pętla *while*

Wielokrotne wykonanie bloku dopóki spełniony jest warunek kontrolujący pętlę:

```
print 'Zgadnij cyfrę od 1 do 9'
```

```
while input() != 7:
```

```
    print 'Zła odpowiedz!'
```

```
    print 'Wracam na początek pętli'
```

```
print 'Trafileś'
```

Pętla for

- Iteracja przy użyciu elementów z listy:

```
print 'Enumeracja liczb 0-10: ',  
for i in [0,1,2,3,4,5,6,7,8,9,10]:  
    print i, ' ',  
print '\nKoniec enumeracji.'
```

- Zmienna i przyjmuje kolejne wartości z listy

Pętla *for*

Wykonanie bloku programu z góry określoną ilość razy

```
print 'Enumeracja liczb 0-99: ',  
for i in range(100):  
    print i, ' ',  
print '\nKoniec enumeracji.'
```

- funkcja range(100) generuje listę liczb w zakresie 1 – 100
- range(start,stop,krok) – range(80, 15, -5)
- type(range(10)) - <type 'list'>

Kontrola pętli

- pominięcie jednego cyklu pętli

continue

- całkowite wyjście z pętli

break

- pusta instrukcja / nie rób nic

pass

Funkcje

Wyodrębnienie części programu z możliwością wywołania z dodatkowymi parametrami:

```
def dodaj(a,b): # definicja funkcji
    c=a+b
    return c    # zwraca wartość do programu

print dodaj(2,3) # początek programu

wyn=dodaj      #funkcje można przypisać do zmiennej
print wyn(3,4)
```

Funkcje rekurencyjne

Przykładowy problem sumowania liczb można rozwiązać iteracyjnie:

```
def liczsume(listaLiczba):  
    suma = 0  
    for liczba in listaLiczba:  
        suma = suma + liczba  
    return suma
```

```
print( liczsume([1,3,6,8,12]) )
```

Funkcje rekurencyjne

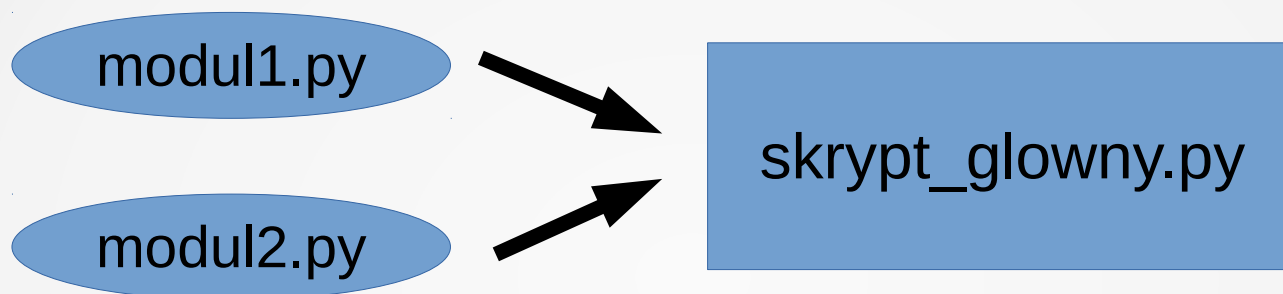
Ten sam problem można rozwiązać rekurencyjnie (ang. *recursive*):

```
def liczsume(listaLiczby):  
    if len(listaLiczby) == 1:  
        return listaLiczby[0]  
    else:  
        return listaLiczby[0] + liczsume(listaLiczby[1:])
```

```
print( liczsume([1,3,6,8,12]) )
```

Moduły

Python pozwala na dzielenie programu na mniejsze pliki



skrypt główny będzie zaczynał się:

```
import modul1
```

```
import modul2
```

Operacje na plikach - IO

Input/Output streams są prostym sposobem na zapisywanie plików:

```
our_file = open( 'JakasNazwa.txt', 'w')
```

```
our_file.write(' Ciąg testowy do zapisania \n')
```

```
our_file.read()                # wczytaj wszystko
```

```
our_file.readline()           # wczytaj linię
```

```
our_file.seek(5)               # przeskocz do 6 znaku
```

```
our_file.read(1)               # czyta „J” z pliku
```

Obsługa wyjątków

W miejscach wrażliwych na nieprzewidziane błędy (np. przy otwieraniu pliku), możemy dodać test na ich wystąpienie używając struktury ***try...except...else***

try:

plik=open('NazwaPliku.txt', w)

except:

print('Błąd dostępu do pliku')

else:

plik.read()

Bez tej struktury w przypadku błędu odczytu/zapisu program zakończył by działanie bezwarunkowo.

Klasy i metody

- klasa jako kolejny poziom abstrakcji

```
class Kalk(object):
```

```
    def __init__(self, s='Inicjalizuje klase kalkulatora'):
```

```
        self.zmie = s # przypisanie wartości do zmiennej lokalnej
```

```
        print self.zmie # wyświetlenie napisu
```

```
    def dodaj(self, a, b)
```

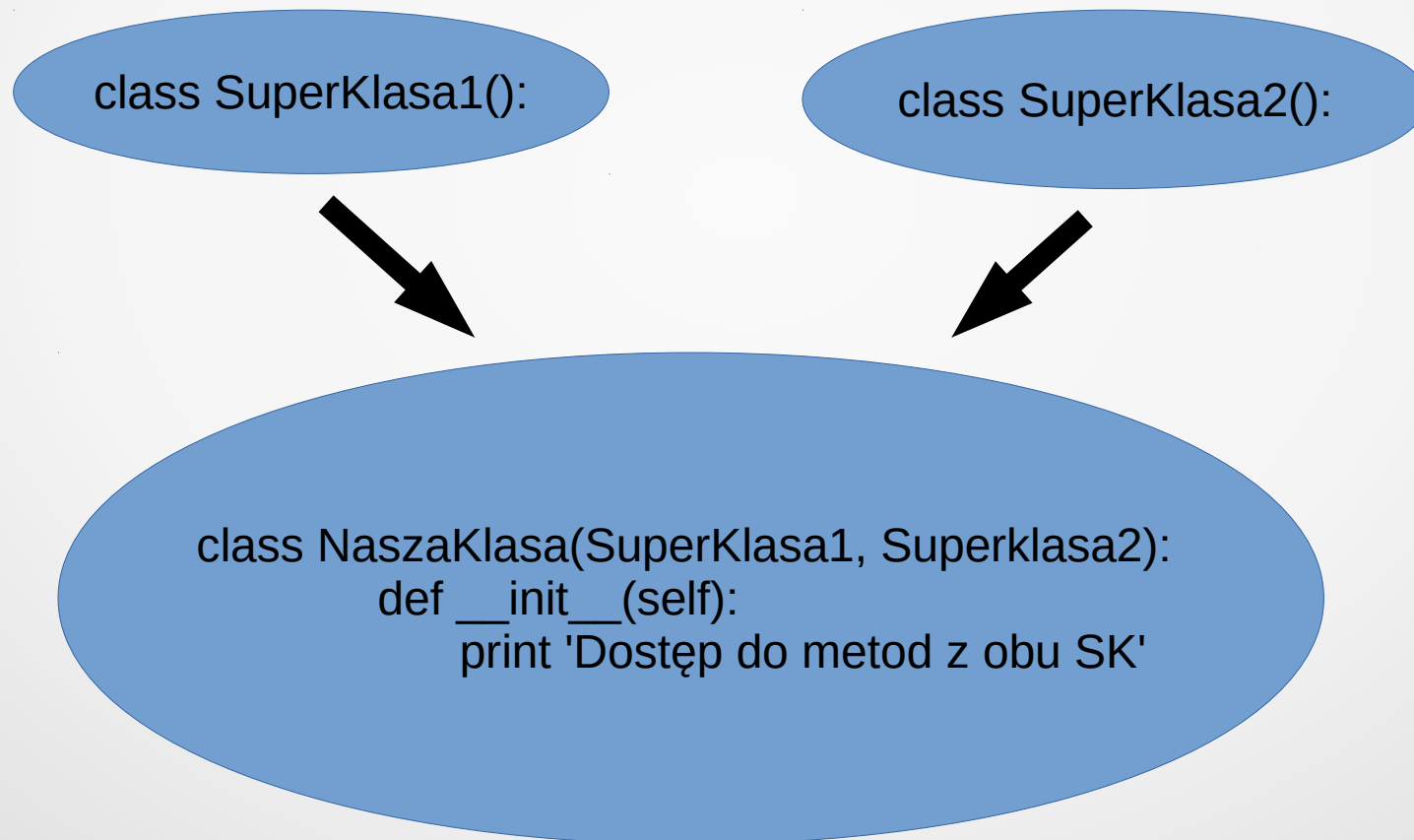
```
        return a+b
```

```
nowyobiekt1=Kalk() # początek programu
```

```
nowyobiekt2=Kalk('Tworze klase z parametrem') # drugi  
obiekt
```

Dziedziczenie klas

Najważniejsza cecha programowania obiektowego to możliwość dziedziczenia z istniejących klas.



Dziedziczenie klas

Przykład dziedziczenia klasy Kalk:

```
class PodKalk(Kalk):  
    def __init__(self):  
        print 'Inicjalizuję podklasę'  
    def odejmij(a, b):  
        return a-b
```

```
nowypodkalk=PodKalk()  
print nowypodkalk.odejmij(8 ,2)  
print nowypodkalk.dodaj(3, 5)
```

Operacje na plikach - JSON

- Standardowe IO jest dobre do prostych danych ale niewygodne przy złożonych:

```
our_data={'par1':'aaa','par2':324.21,'par3':13}
```

- Moglibyśmy iterować po danych, lecz jest to skomplikowane. Ale możemy użyć biblioteki JSON:

```
import json
```

```
our_file=open('file_with_our_data.json','w')
```

```
json.dump(our_data, our_file)
```

```
our_file.close()
```

```
our_file=open('file_with_our_data.json','r')
```

```
print json.load(our_file)
```

- Wyjście będzie wyglądać:

```
{u'par3': 13, u'par2': 324.21, u'par1': u'aaa'}
```

Menadżer kontekstu **with**...

- Nasz plik zawiera współrzędne:

```
101 -28750.00 296000.00
102 -28750.00 295950.00
103 -28750.00 295900.00
```

- Możemy użyć **with**, żeby iterować po liniach w naszym pliku, zapewnia nas to, że plik będzie zamknięty po zapisaniu:

```
our_table=[]                # pusta lista/tablica współrzędnych
with open('OurFile.txt','r') as our_file:    # otwarcie pliku
    for linia in plik:                # iteracja
        nr, x, y = linia.split()      # rozdzielenie ciągu
        our_table.append( [int(nr), float(x), float(y)] )
```

Powyższy kod wczyta listę współrzędnych.

Biblioteki standardowe

- możemy korzystać z wielu bibliotek standardowych, które zawierają często używane funkcje

import math, sys

- zamiast całej biblioteki możemy importować wybrane klasy

from PyQt4 import *

from PyQt4 import QWidget

- możemy sprawdzić jakie funkcje zostały zaimportowane

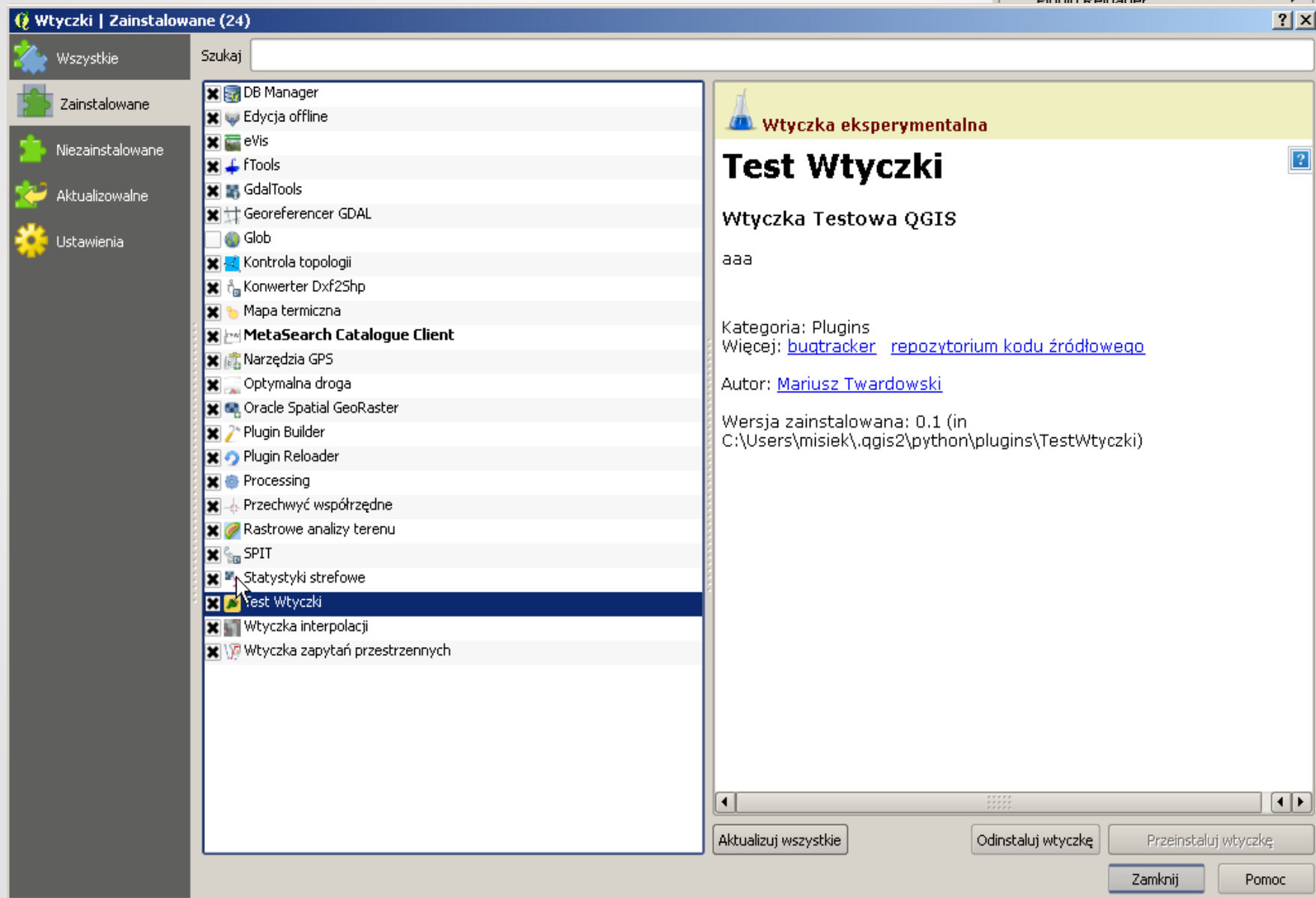
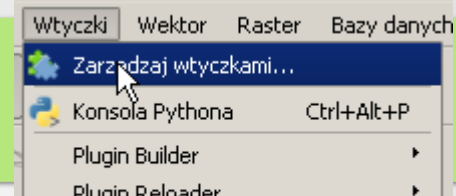
dir(math)

Python w QGIS

- integracja Python i PyQt w instalacji
- środowisko uruchomienia OSGeo4W
- realizacja wtyczki dla QGis
- automatyczne generowanie kodu
- instalacja wtyczki w programie
- katalog wtyczek

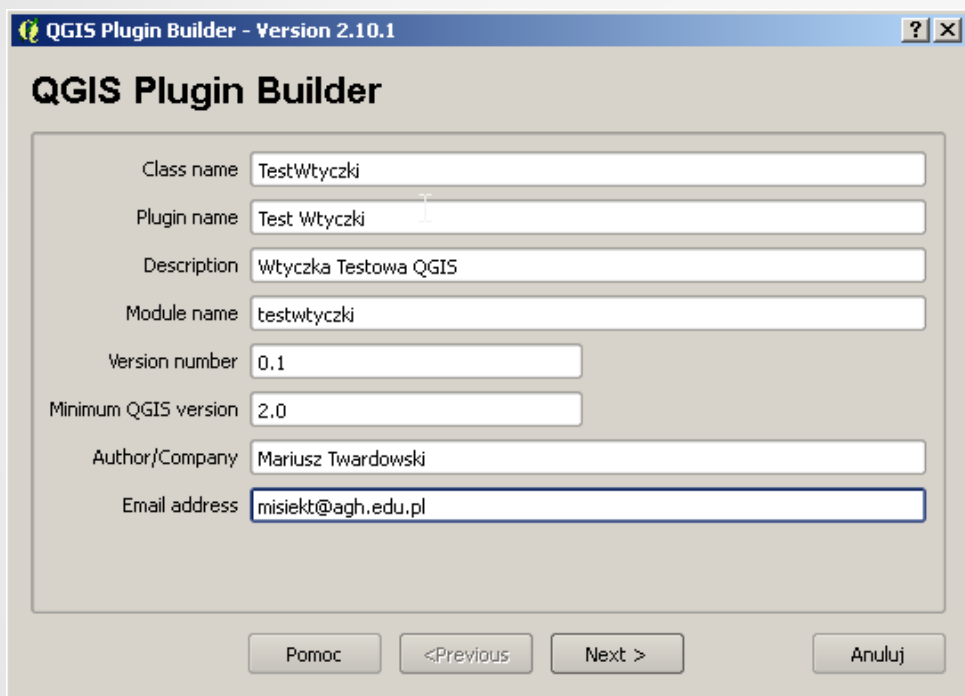
`%USERPROFILE%/.qgis2/python/plugins`

Zarządzanie wtyczkami



Generowanie bazy wtyczki

- wtyczka Plugin Builder



QGIS Plugin Builder - Version 2.10.1

QGIS Plugin Builder

Class name: TestWtyczki

Plugin name: Test Wtyczki

Description: Wtyczka Testowa QGIS

Module name: testwtyczki

Version number: 0.1

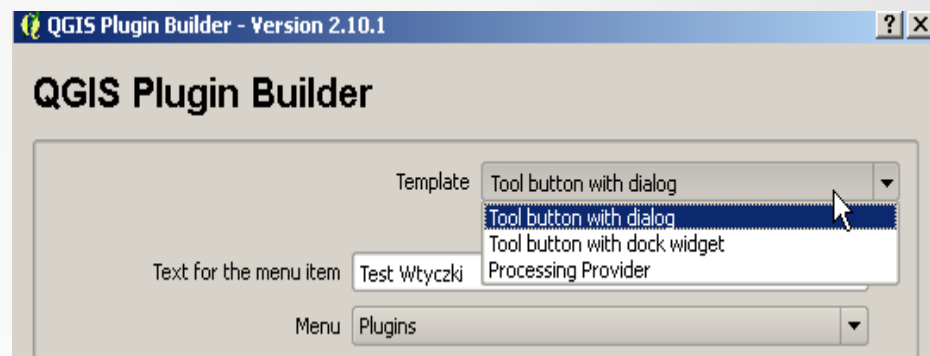
Minimum QGIS version: 2.0

Author/Company: Mariusz Twardowski

Email address: misiekt@agh.edu.pl

Pomoc <Previous Next > Anuluj

- translatable pyuic4 i pyrcc4
- wtyczka Plugin Reloader



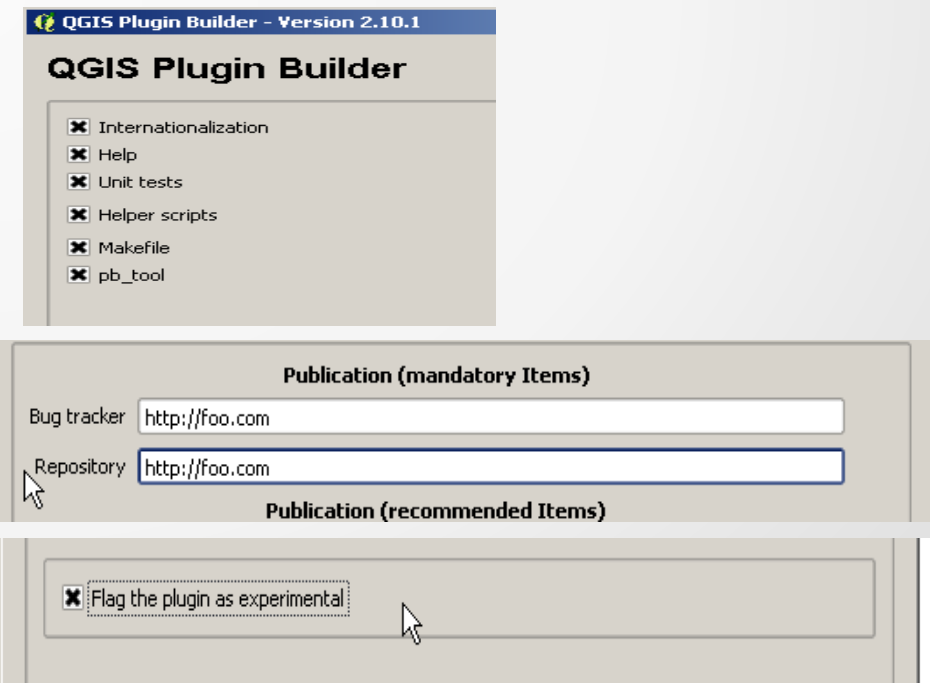
QGIS Plugin Builder - Version 2.10.1

QGIS Plugin Builder

Template: Tool button with dialog (selected), Tool button with dock widget, Processing Provider

Text for the menu item: Test Wtyczki

Menu: Plugins



QGIS Plugin Builder - Version 2.10.1

QGIS Plugin Builder

☒ Internationalization

☒ Help

☒ Unit tests

☒ Helper scripts

☒ Makefile

☒ pb_tool

Publication (mandatory Items)

Bug tracker: http://foo.com

Repository: http://foo.com

Publication (recommended Items)

☒ Flag the plugin as experimental

Zawartość plików wtyczki

- inicjalizacja w pliku `__init__.py`

```
def classFactory(iface):  
    from .testwtyczki import TestWtyczki  
    return TestWtyczki(iface)
```

- plik klasy `testwtyczki.py`

```
import resources  
  
from testwtyczki_dialog import TestWtyczkiDialog  
  
class Testwtyczki:  
    (...)   
    self.dlg=TestWtyczkiDialog()
```

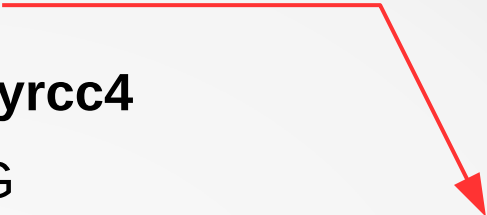
Zawartość plików

- definicja ikony w pliku **resources.qrc**
- ikona wtyczki a plik **resources_rc.py** - **pyrcc4**
- reprezentacja szesnastkowa obrazu PNG



```
qt_resource_data = "\
\x00\x00\x04\x0a\
\x89\
```

```
\x50\x4e\x47\x0d\x0a\x1a\x0a\x00\x00\x00\x0d\x49\x48\x44\x52\x00\
\x00\x00\x17\x00\x00\x00\x18\x08\x06\x00\x00\x00\x11\x7c\x66\x75\
\x00\x00\x00\x01\x73\x52\x47\x42\x00\xae\xce\x1c\xe9\x00\x00\x00\
\x06\x62\x4b\x47\x44\x00\xff\x00\xff\x00\xff\xa0\ (....) ,,
```



```
<RCC>
  <qresource prefix="/plugins/TestWtyczki"
  >
    <file>icon.png</file>
  </qresource>
</RCC>
```

Zawartość plików

- inicjalizacja klasy dialogu **testwtyczki_dialog.py** z automatyczną translacją interfejsu

```
FORM_CLASS, _ = uic.loadUiType(os.path.join(  
    os.path.dirname(__file__), 'testwtyczki_dialog_base.ui'))
```

```
class TestWtyczkiDialog(QtGui.QDialog, FORM_CLASS):
```

```
    def __init__(self):
```

```
        self.setupUi(self)
```

- tradycyjna inicjalizacja klasy dialogu **testwtyczki_dialog.py**

```
from ui_testwtyczki import Ui_TestWtyczkiDialogBase
```

```
class TestWtyczkiDialog(QtGui.QDialog, Ui_TestWtyczkiDialogBase):
```

```
    def __init__(self):
```

```
        self.setupUi(self)
```

Zawartość plików

- ui_testwtyczki.py - translacja testwtyczki_dialog_base.ui w pyuic4

-*- coding: utf-8 -*-

```
from PyQt4 import QtCore, QtGui
```

```
class Ui_TestWtyczkiDialogBase(object):
```

```
    def setupUi(self, TestWtyczkiDialogBase):
```

```
        TestWtyczkiDialogBase.setObjectName("TestWtyczkiDialogBase")
```

```
        TestWtyczkiDialogBase.resize(400, 300)
```

```
        self.button_box = QtGui.QDialogButtonBox(TestWtyczkiDialogBase)
```

```
        self.button_box.setGeometry(QtCore.QRect(30, 240, 341, 32))
```

```
        self.button_box.setOrientation(QtCore.Qt.Horizontal)
```

```
        self.button_box.setStandardButtons(QtGui.QDialogButtonBox.Cancel|QtGui.QDialogButtonBox.Ok)
```

```
        self.button_box.setObjectName(_fromUtf8("button_box"))
```

```
        self.retranslateUi(TestWtyczkiDialogBase)
```

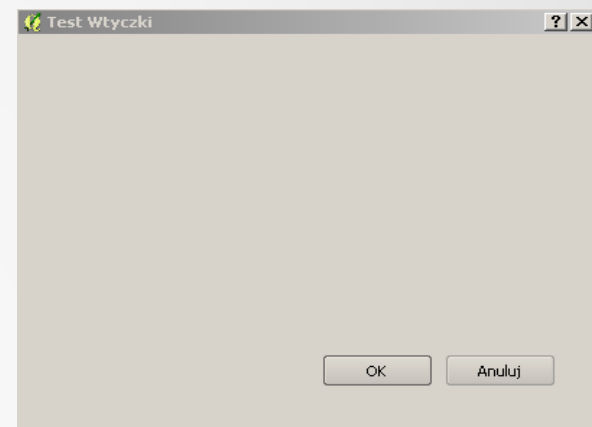
```
        QtCore.QObject.connect(self.button_box, QtCore.SIGNAL(("accepted()")), TestWtyczkiDialogBase.accept)
```

```
        QtCore.QObject.connect(self.button_box, QtCore.SIGNAL(("rejected()")), TestWtyczkiDialogBase.reject)
```

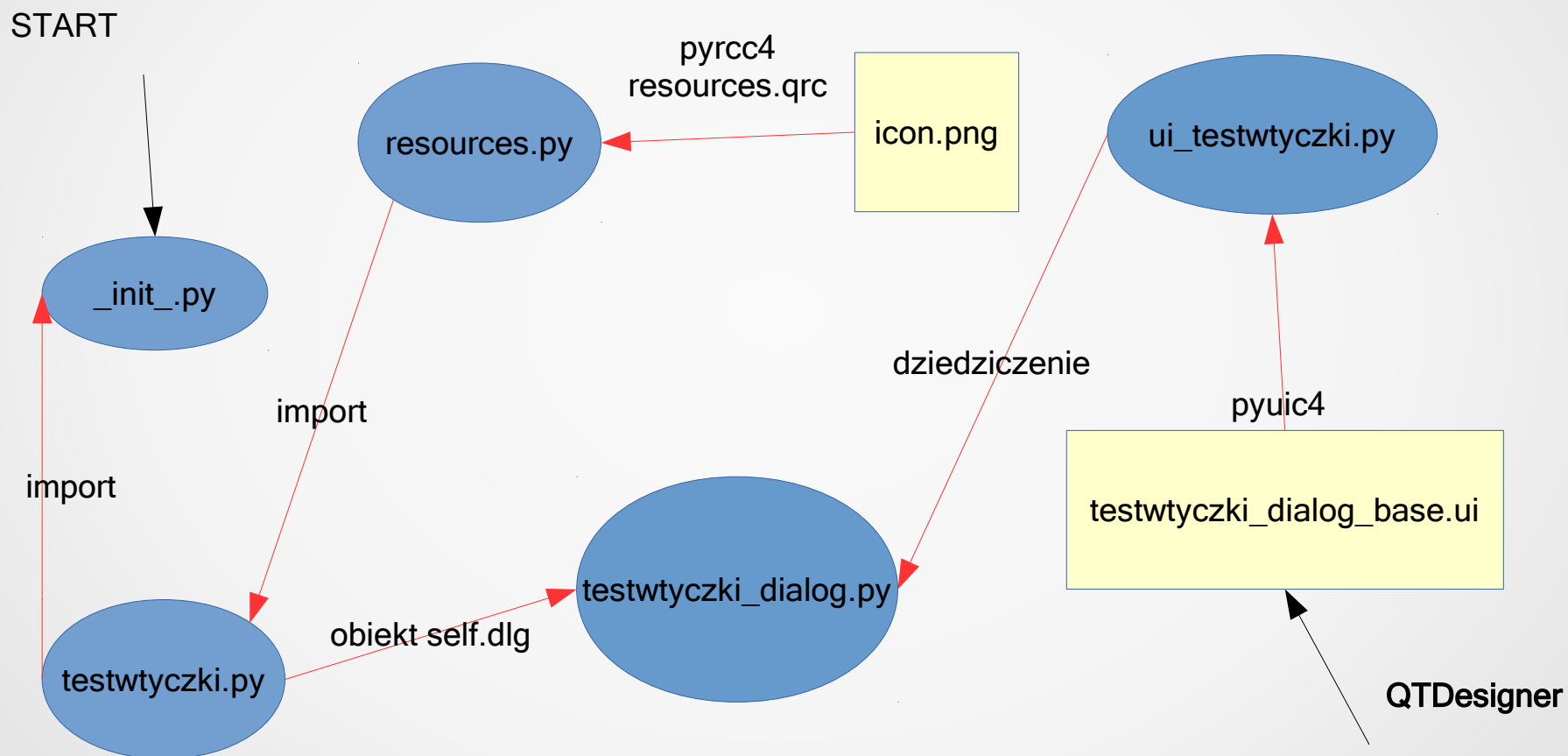
```
        QtCore.QMetaObject.connectSlotsByName(TestWtyczkiDialogBase)
```

```
def retranslateUi(self, TestWtyczkiDialogBase):
```

```
    TestWtyczkiDialogBase.setWindowTitle(_translate("TestWtyczkiDialogBase", "Test Wtyczki", None))
```

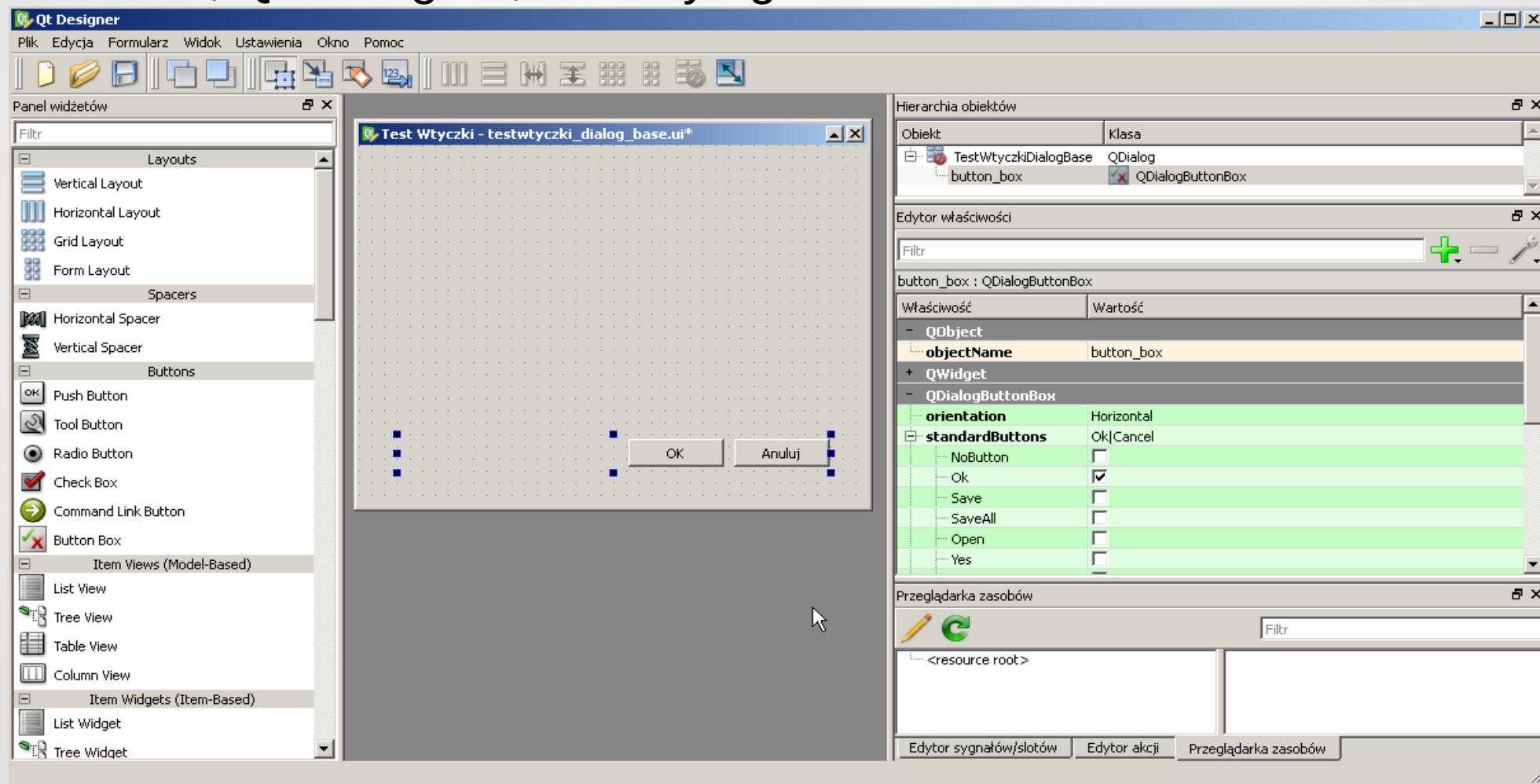


Schemat programu



Interfejs wtyczki

- RAD, QTDesigner, widżety i generowanie kodu



- QTDesigner → plik .UI → pyuic4 → plik .PY

Sygnały i sloty w PyQt4

- element interfejsu wysyła sygnał

clicked() pushed() released()

- metoda connect() łączy sygnał ze slotem (metodą)

self.bPrzycisk.clicked.connect(self.naszaMetoda)

- połączenie automatyczne **connectSlotsByName()**

on_bPrzycisk_clicked()

on_bPrzycisk_pushed()

on_bPrzycisk_released()

Funkcjonalność programu

- transformacja współrzędnych KUL do EPSG:2178

$$xw = xbw + a1 + a3*x - a4*y + a5*(x*x - y*y) - 2*a6*x*y$$

$$yw = ybw + a2 + a3*y + a4*x + 2*a5*x*y + a6*(x*x - y*y)$$

- współczynniki

$$xbp = -3.04419376190476 \times 10^4 \quad ybp = 2.89030878571429 \times 10^5$$

$$xbw = 5.54659313500000 \times 10^6 \quad ybw = 7.42852359576190 \times 10^6$$

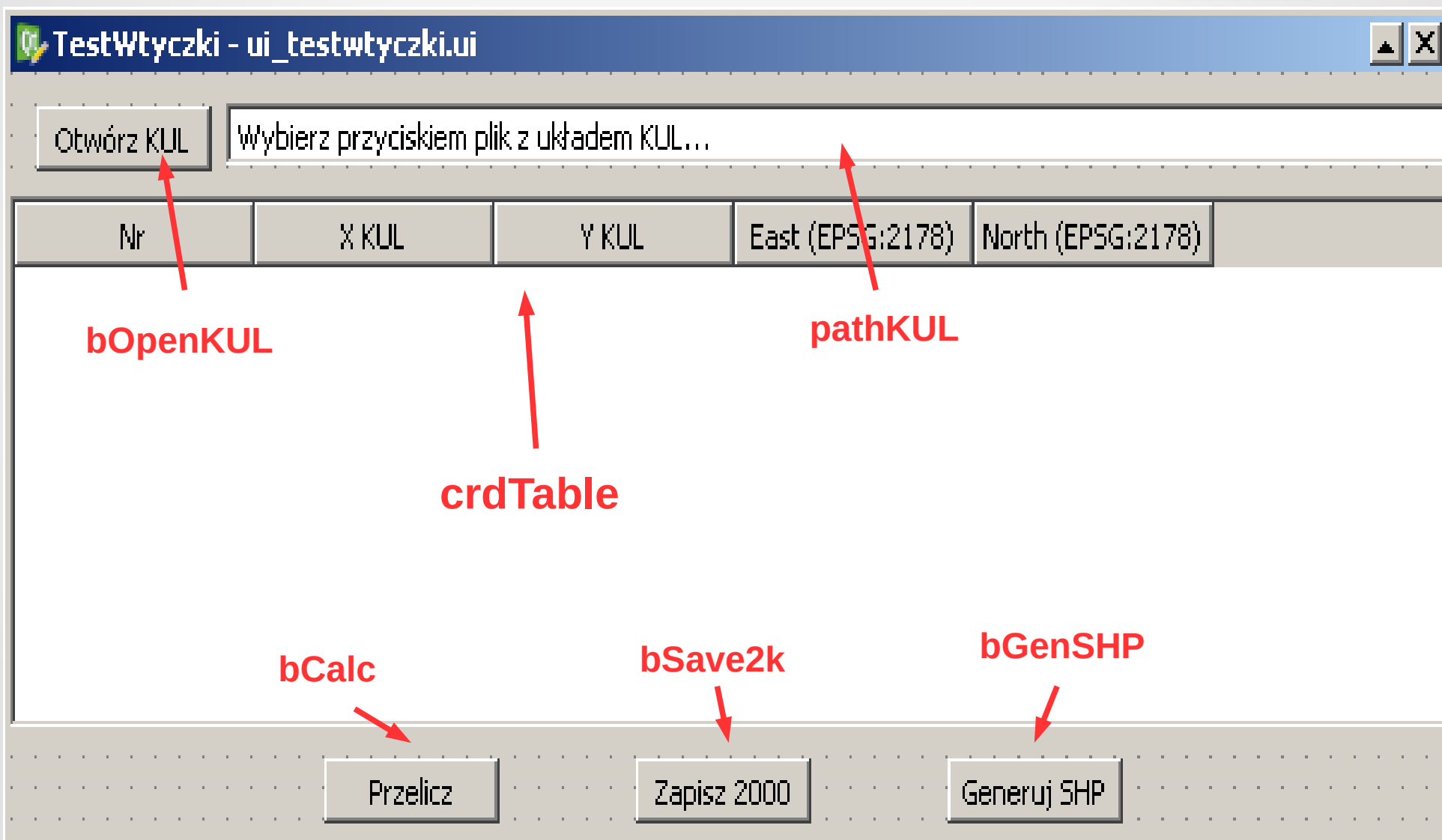
$$x = x - xbp \quad y = y - ybp$$

$$a1 = 2.73660450713879 \times 10^{-22} \quad a2 = 7.69215691432381 \times 10^{-2}$$

$$a3 = -9.99116225304304 \times 10^{-1} \quad a4 = 4.05478646361601 \times 10^{-2}$$

$$a5 = 6.77548043232547 \times 10^{-11} \quad a6 = 9.08724778098026 \times 10^{-10}$$

Interfejs programu



Implementacja kodu

- otwarcie pliku KUL - **on_bOpenKUL_released()**
- funkcja przeliczenia współrzędnych - **calcK2k()**
- obsługa przycisku przelicznia - **on_bCalc_released()**
- zapisanie pliku wyjściowego - **on_bSave2000()**
- tworzenie pliku graficznego SHP - **on_bGenSHP()**

Samodzielna aplikacja

- Interfejs można uruchomić bez QGIS

testwtyczki_dialog.py:

```
if __name__ == "__main__":  
    import sys  
    app = QtGui.QApplication(sys.argv)  
    myapp = TestWtyczkiDialog()  
    myapp.show()  
    sys.exit(app.exec_())
```

Implementacja kalkulatora PyQt4

- użycie QTDesigner do stworzenia interfejsu
widżety QT:

Line Edit

Push Button

Label



- implementacja metod dla działań
on_bDodaj_released(), on_bMinus_released()....
- komentarze do kodu, wysyłanie przez formularz zaliczeń

Operacje na rastrach - PIL

- Użycie standardowej biblioteki Pillow

```
from PIL import Image      # import klasy Image z modułu PIL
import webbrowser          # import modułu webbrowser
```

```
obraz=Image.open('test.jpg')
print obraz.format, obraz.size, obraz.mode # właściwości obrazu
#obraz.show()      # w wersji QGIS 2.10 nie działa domyslnie na Windows 7
obraz.save('test.tif') # zapis obrazu jako tif
```

```
obraz45=obraz.rotate(45)      # obrót obrazu o 45 stopni
obraz45.save('test45.jpg')    # zapis obróconego obrazu
obraz.rotate(45).save('test45.tif') # obrót i zapis w jednej lini
```

```
webbrowser.open('test.tif')   # zamiast obraz.show() używamy webbrowser.
webbrowser.open('test45.jpg')
```

Operacje na rastrach - PIL

- Inne metody klasy Image:

<code>obraz.convert('1').save('testbw.tif')</code>	<code>#konwersja do 1-bit BW</code>
<code>obraz.convert('L').save('testgray.tif')</code>	<code>#odcienie szarosci 8bit</code>
<code>obraz.convert('P').save('testpal.tif')</code>	<code>#paleta 8-bit</code>
<code>obraz.crop([100,100,200,200]).save('testcrop.tif')</code>	<code>#wycinanie tupla [x0,y0,x1,y1]</code>
<code>obraz.resize([100,100]).save('testresize.tif')</code>	<code>#zmiana rozmiaru tupla [w,h]</code>

- Biblioteka pozwala na rozkład obrazu RGB i generowanie

histogramu:

<code>R, G, B = obraz.split()</code>	<code>#rozkłada obraz na składowe tuple</code>
<code>bgr=Image.merge("RGB",[B,G,R])</code>	<code>#składa obraz z zamienioną kolejnością</code>
<code>hist=bgr.histogram()</code>	<code># tworzy histogram - lista połączona BGR</code>
<code>histB=B.histogram()</code>	<code># histogram B</code>
<code>print histB, '\n', hist</code>	
<code>bgr.save('testbgr.tif')</code>	
<code>webbrowser.open('testbgr.tif')</code>	

Operacje na rastrach - PIL

- Tworzenie nowego obrazu i operacje na pixelach

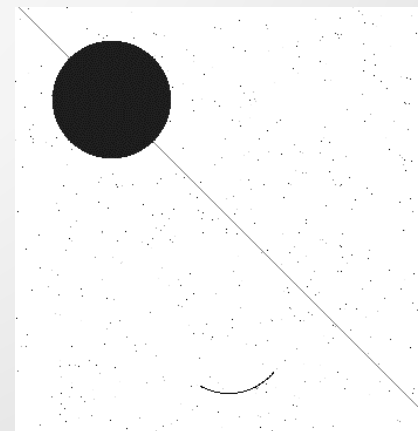
```
import random
obr=Image.new("L",[512,512],'white')      #utwórz pusty obraz new(mode, size,color)
obrpx=obr.load()                          #dostęp do pixeli obrpx[x,y]
for pixcount in range(1000):
    rnd=random.randint                    #skrót funkcji random.randint
    x, y, c=rnd(0,511), rnd(0,511), rnd(0,255) #wygeneruj losowo x,y,c
    obrpx[x,y]=c                         #przypisz losowe wartości dla pixeli
obr.save('testrnd.tif')                  #można użyć metody putpixel
```

- Druga klasa PIL to ImageDraw, pozwala rysować:

```
import ImageDraw

rys=ImageDraw.Draw(obr)
rys.line( (0,0) + obr.size, fill=128)
rys.ellipse( (100,100,200,200), fill=30 )
rys.arc((200,300,300,400), 40, 120)

obr.save('testdraw.tif')
```



Wizualizacja - matplotlib

- Biblioteka matplotlib umożliwia łatwe wyświetlanie zarówno rastrów jak i wykresów/wektorów:

```
# coding=1250
import matplotlib.pyplot as plt #zaimportuj pyplot pod nazwą plt

obraz = plt.imread('test.jpg') #wczytaj obraz RGB
fig1=plt.imshow(obraz)         #dodaj obraz do kontekstu

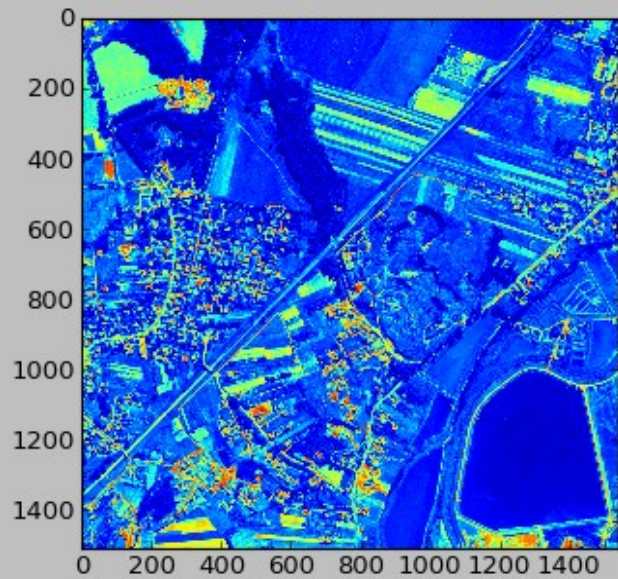
#stwórz nowy kontekst i 6 podkontekstów
fig2, ((rx1, gx1, bx1), (rx2, gx2, bx2)) = plt.subplots(2, 3)

obrR,obrG,obrB =obraz[:, :,0], obraz[:, :,1], obraz[:, :,2] #wyodrębnij kanały

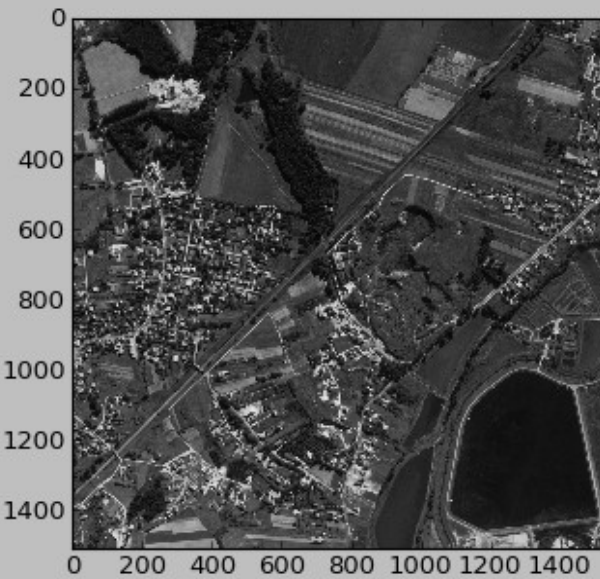
rx1.imshow(obrR)                #wyświetl kanał-pseudokolor
rx2.hist(obrR.ravel(), 256, [0,256]) #wyświetl histogram
gx1.imshow(obrG,cmap='gray')    #wyswietl kanał-odcienie
gx2.hist(obrG.ravel(), 256, [0,256])
bx1.imshow(obrB,clim=(20,200))
bx2.hist(obrB.ravel(), 256, [0,256])
plt.show()                      #wyświetl konteksty graficzne
```



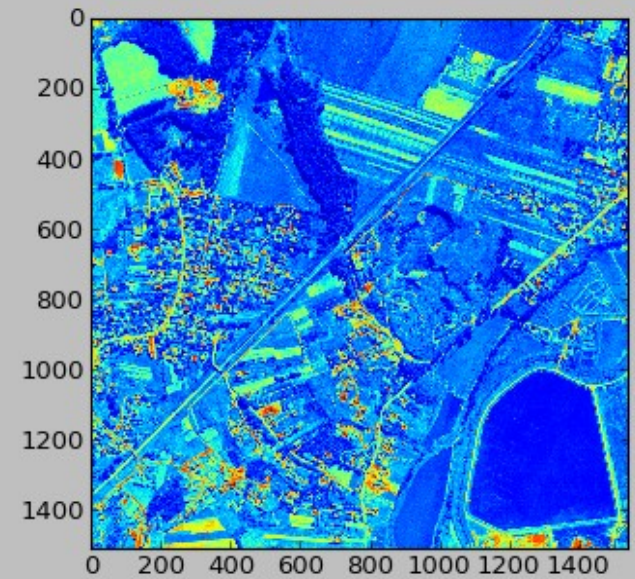
matplotlib - subplots



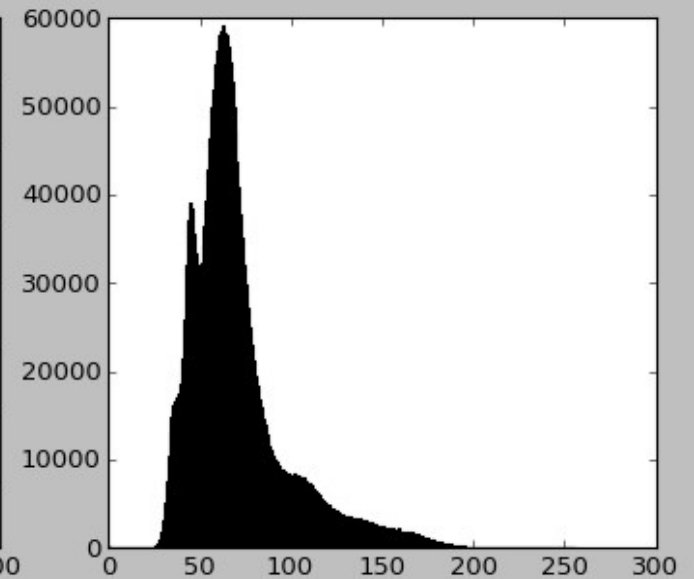
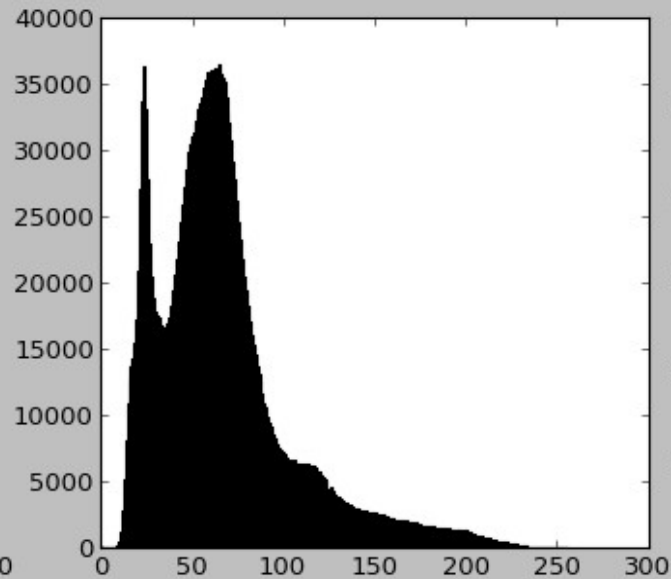
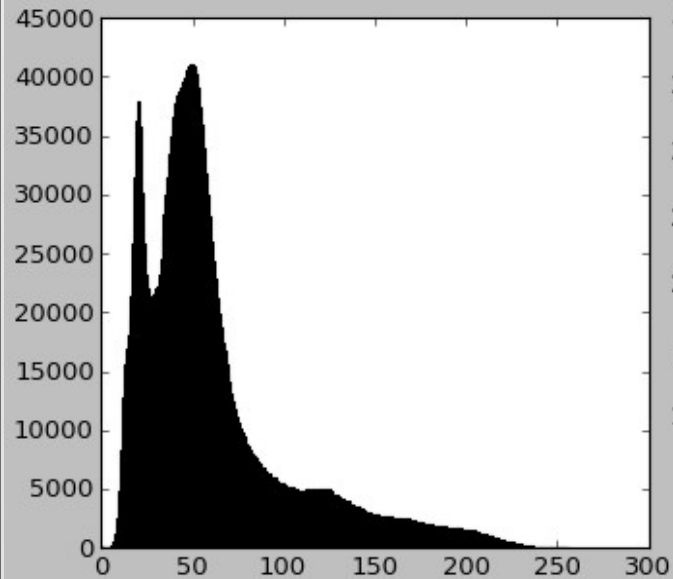
obrR



obrG



obrB

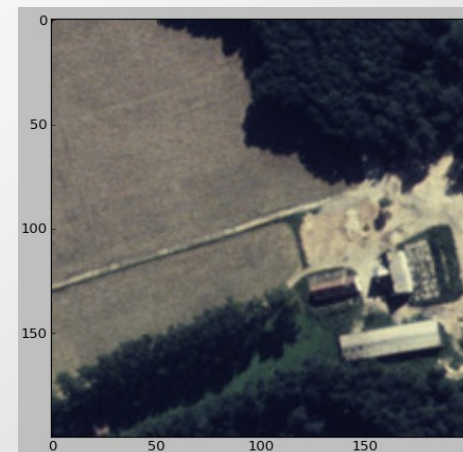


PIL+matplotlib

- Biblioteka matplotlib pomaga w wizualizacji ale ma ubogi zakres funkcji przetwarzania obrazu. Możemy połączyć funkcjonalność z Image:

```
from PIL import Image
import matplotlib.image as mimg #zaimportuj obie biblioteki
```

```
iobraz = Image.fromarray(obraz) #skonwertuj obraz do macierzy Image
print iobraz.format, iobraz.size, iobraz.mode #wyświetl informacje o obrazie
iobraz=iobraz.crop([100,100,300,300]) #przytnij obraz w Image
obraz=mimg.pil_to_array(iobraz) #skonwertuj z powrotem do MPL
plt.figure() #dodaj nowy kontekst MPL
fig5=plt.imshow(obraz) #dodaj obraz do kontekstu
plt.show() #wyświetl używając MPL
```



Łączenie QT i matplotlib

- W Notepad++....