



# Język Python na potrzeby GIS

Katedra Geoinformacji, Fotogrametrii i Teledetekcji Środowiska

©2014 Mariusz Twardowski

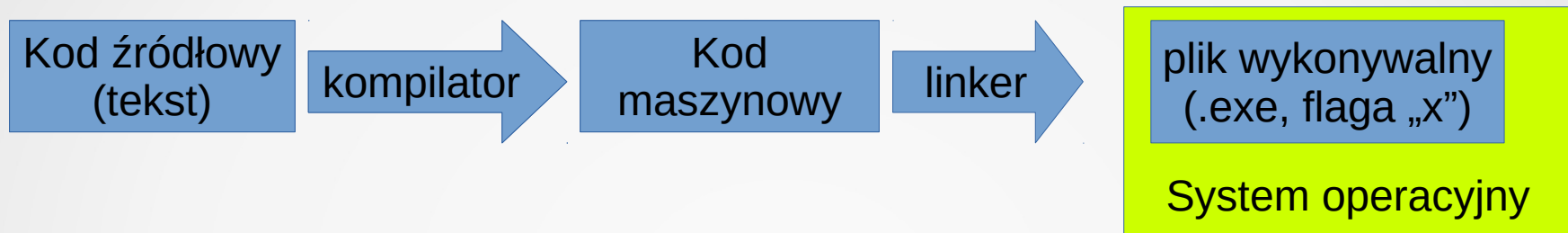
LibreOffice Impress  
<http://libreoffice.org>

# 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.

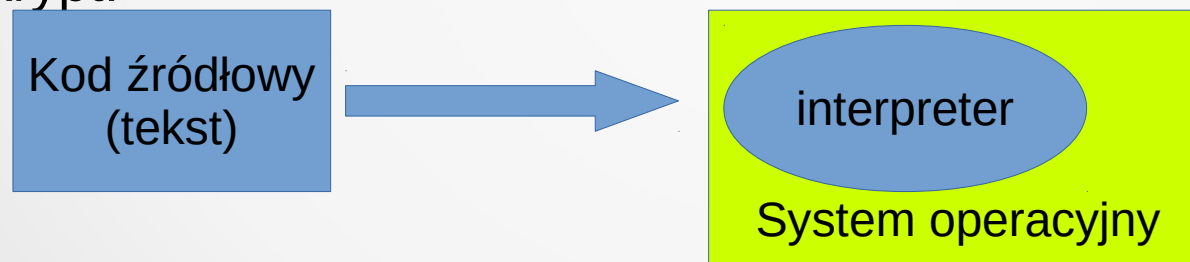
# Program vs skrypt

- Program:



zalety – szybkość i spójność wykonania, pamięć

- Skrypt:



zalety – wieloplatformowość, łatwe testowanie kodu

# 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.

# Środowisko uruchomienia

- QGIS 2.4.0 – <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

# 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.

# Podstawy języka Python

- dokumentacja:

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

- komentarz:

znak #

- każda wartość jest obiektem
- konsola Pythona – pojedyncze wyrażenia
- skrypt – połączenie wielu poleceń

# 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)



# Podstawowe operatory

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

# 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

$1 \text{ and } 1 = 1$ ,  $1 \text{ and } 0 = 0$ ,  $0 \text{ and } 1 = 0$ ,  $0 \text{ and } 0 = 0$

$1 \text{ or } 1 = 1$ ,  $1 \text{ or } 0 = 1$ ,  $0 \text{ or } 1 = 1$ ,  $0 \text{ or } 0 = 0$

$\text{not } 1 = 0$ ,  $\text{not } 0 = 1$

- łączenie warunków

$10 == 10 \text{ or } 10 >= 3$  # prawda, jeden warunek spełniony

$3 < 4 \text{ and } 4 < 10$  # prawda, oba warunki spełnione

$3 < 4 < 10$  # uproszczenie zapisu

# 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

# 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'  
print 'Trafileś'
```



# 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()` generuje liczby w zakresie 1 - 100

# Kontrola pętli

- pominięcie jednego cyklu pętli

**continue**

- całkowite wyjście z pętli

**break**

- pusta instrukcja / nie rób nic

**pass**

# Funkcje

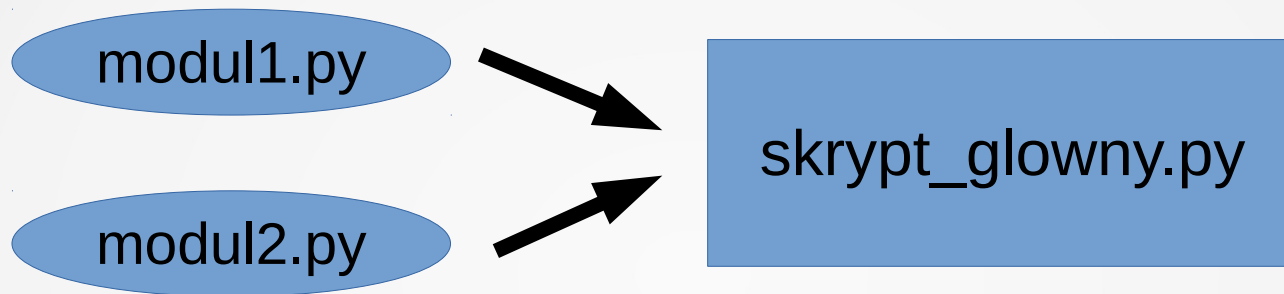
Wyodrębnienie części programu z możliwością wywołania z dodatkowymi parametrami:

```
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)
```

# Moduły

Python pozwala na dzielenie programu na mniejsze pliki



skrypt główny będzie zaczynał się:

```
import modul1
```

```
import modul2
```

# Klasy i metody

- klasa jako kolejny poziom abstrakcji

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

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

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

```
        print self.zaps # wyswietlenie 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 korzystania z już istniejących klas.

```
class SuperKlasa1():
```

```
class SuperKlasa2():
```

```
class NaszaKlasa(SuperKlasa1, SuperKlasa2):  
    def __init__(self):  
        print 'Dostęp do metod z obu SK'
```

# 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)
```

# 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.



# 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 QtWidgets
```

- możemy sprawdzić jakie funkcje zostały zaimportowane

```
dir(math)
```

# 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(listaLiczb):
```

```
    if len(listaLiczb) == 1:
```

```
        return listaLiczb[0]
```

```
    else:
```

```
        return listaLiczb[0] + liczsume(listaLiczb[1:])
```

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

# 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%/.qgis/python/plugins`

# Generowanie bazy wtyczki

- zarządzanie wtyczkami w QGIS
- wtyczka Plugin Builder



The image shows a screenshot of the 'QGIS Plugin Builder' dialog box. It contains several input fields for configuring a plugin. The fields are: 'Class name' (TestWtyczki), 'Plugin name' (Test Wtyczki), 'Description' (Wtyczka testowa Quantum Gis), 'Version number' (0.1), 'Minimum QGIS version' (2.0), 'Text for the menu item' (Wtyczka Testowa), 'Author/Company' (Mariusz Twardowski), and 'Email address' (misiect@agh.edu.pl). There is also a checkbox labeled 'Flag the plugin as experimental' which is checked. At the bottom, there are three buttons: 'OK', 'Anuluj', and 'Pomoc'.

Field	Value
Class name	TestWtyczki
Plugin name	Test Wtyczki
Description	Wtyczka testowa Quantum Gis
Version number	0.1
Minimum QGIS version	2.0
Text for the menu item	Wtyczka Testowa
Author/Company	Mariusz Twardowski
Email address	misiect@agh.edu.pl

- translatory **pyuic4** i **pyrcc4**
- wtyczka Plugin Reloader

# Zawartość plików wtyczki

- inicjalizacja w pliku `__init__.py`

```
def classFactory(iface):
```

```
    from testwtyczki import TestWtyczki
```

```
    return TestWtyczki(iface)
```

- plik klasy `testwtyczki.py`

```
import resources_rc
```

```
from testwtyczkidialog import TestWtyczkiDialog
```

```
class Testwtyczki:
```

```
    (...)
```

```
    self.dlg=TestWtyczkiDialog()
```

# Zawartość plików

- ikona wtyczki a plik `resources_rc.py`
- 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\ (...)
```

# Zawartość plików

- inicjalizacja klasy dialogu **testwtyczki/dialog.py**

```
from ui_testwtyczki import Ui_TestWtyczki
```

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

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

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



# Zawartość plików

## • ui\_testwtyczki.py ....

```
class Ui_TestWtyczki(object):
```

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

```
        TestWtyczki.setObjectName(_fromUtf8("TestWtyczki"))
```

```
        TestWtyczki.resize(600, 250)
```

```
        self.bOpenKUL = QtGui.QPushButton(TestWtyczki)
```

```
        self.bOpenKUL.setGeometry(QtCore.QRect(10, 10, 75, 23))
```

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

```
        self.crdTable = QtGui.QTableWidget(TestWtyczki)
```

```
        self.crdTable.setGeometry(QtCore.QRect(0, 40, 601, 171))
```

```
        self.crdTable.setLayoutDirection(QtCore.Qt.LeftToRight)
```

```
        self.crdTable.setAutoFillBackground(False)
```

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

```
        self.crdTable.setColumnCount(5)
```

```
        self.crdTable.setRowCount(0)
```

```
        item = QtGui.QTableWidgetItem()
```

```
        self.crdTable.setHorizontalHeaderItem(0, item)
```

```
        item = QtGui.QTableWidgetItem()
```

```
        self.crdTable.setHorizontalHeaderItem(1, item)
```

```
        item = QtGui.QTableWidgetItem()
```

```
        self.crdTable.setHorizontalHeaderItem(2, item)
```

```
        item = QtGui.QTableWidgetItem()
```

```
        self.crdTable.setHorizontalHeaderItem(3, item)
```

```
        item = QtGui.QTableWidgetItem()
```

```
        self.crdTable.setHorizontalHeaderItem(4, item)
```

```
        self.bSave2000 = QtGui.QPushButton(TestWtyczki)
```

```
        self.bSave2000.setGeometry(QtCore.QRect(260, 220, 75, 23))
```

```
        self.bSave2000.setObjectName(_fromUtf8("bSave2000"))
```

```
        self.bCalc = QtGui.QPushButton(TestWtyczki)
```

```
        self.bCalc.setGeometry(QtCore.QRect(130, 220, 75, 23))
```

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

```
        self.pathKUL = QtGui.QLineEdit(TestWtyczki)
```

```
        self.pathKUL.setGeometry(QtCore.QRect(90, 10, 511, 20))
```

```
        self.pathKUL.setReadOnly(True)
```

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

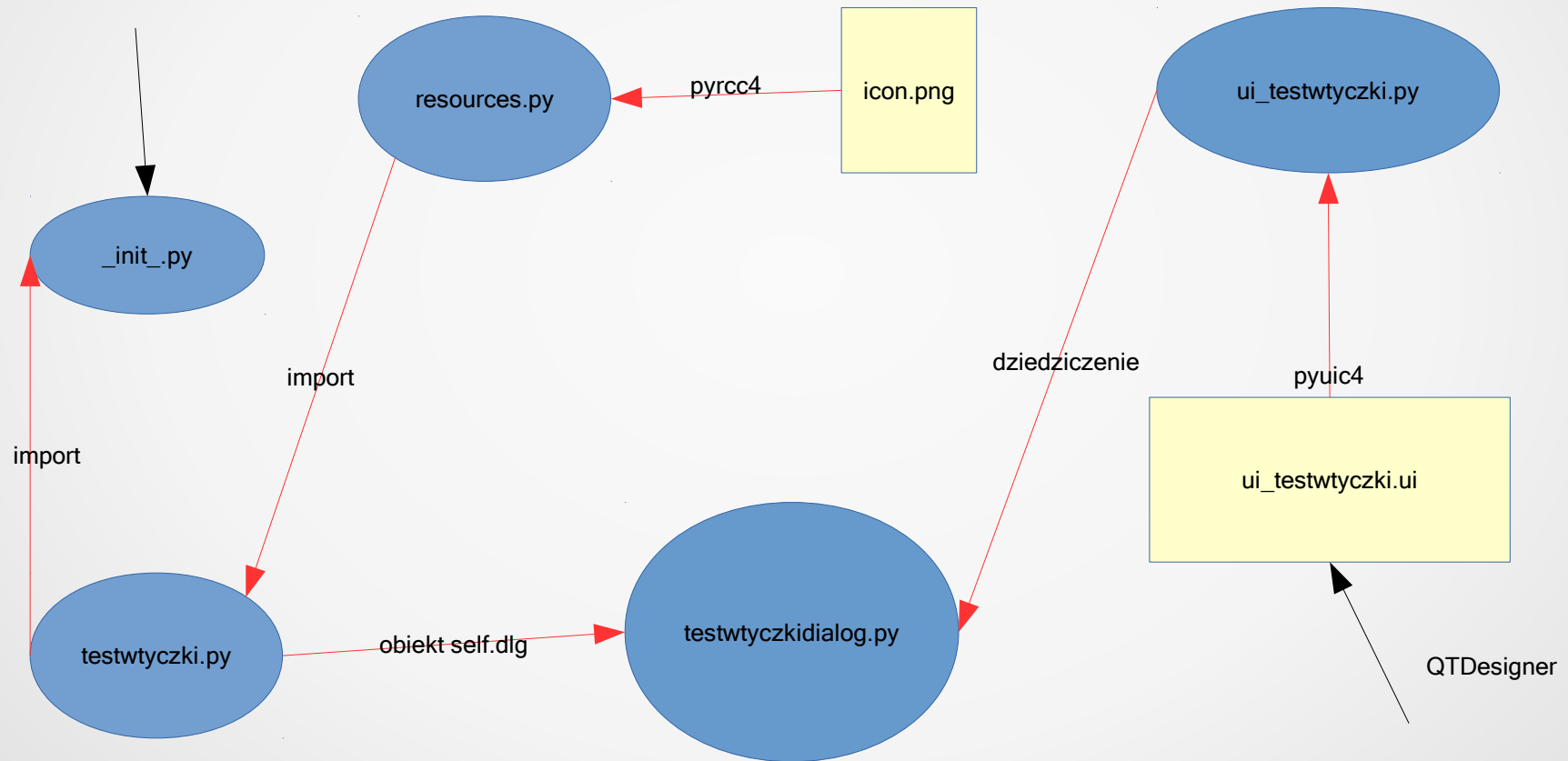
```
        self.bGenSHP = QtGui.QPushButton(TestWtyczki)
```

```
        self.bGenSHP.setGeometry(QtCore.QRect(390, 220, 75, 23))
```

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

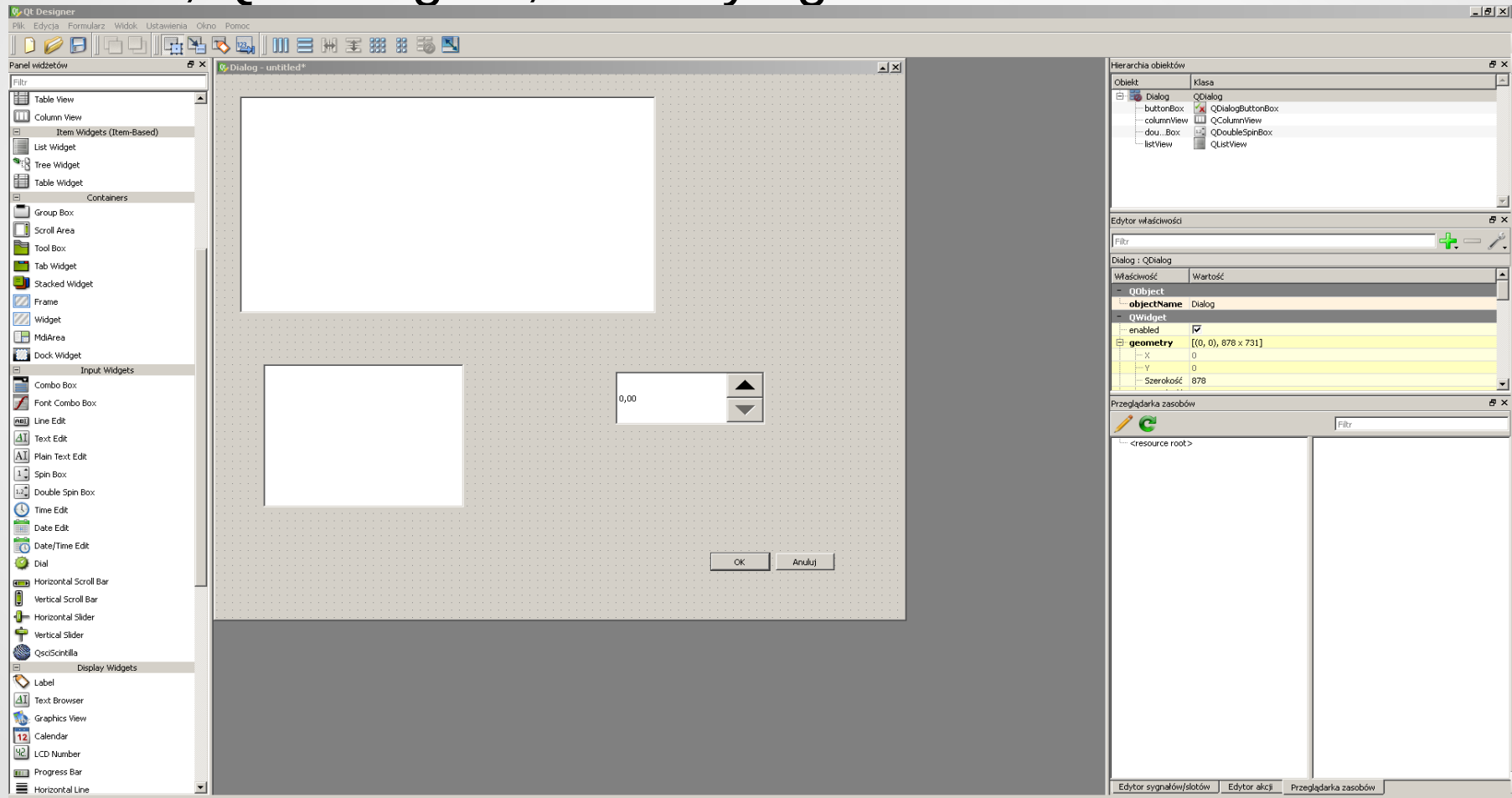
# Schemat programu

START



# 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

$$x_{bp} = -3.04419376190476 \times 10^4 \quad y_{bp} = 2.89030878571429 \times 10^5$$

$$x_{bw} = 5.54659313500000 \times 10^6 \quad y_{bw} = 7.42852359576190 \times 10^6$$

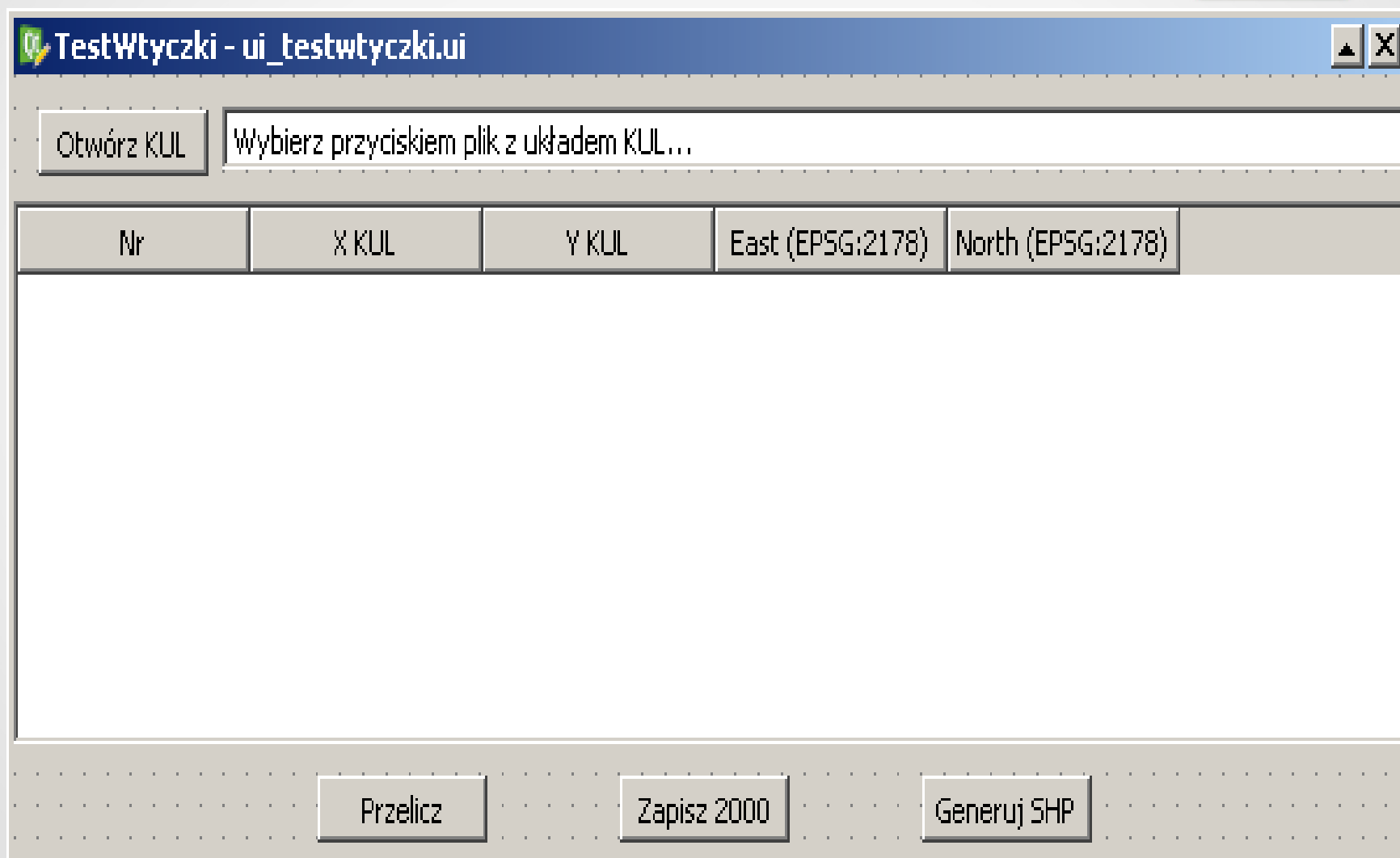
$$x = x - x_{bp} \quad y = y - y_{bp}$$

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

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

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

# 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()**

# Implementacja kalkulatora PyQt4

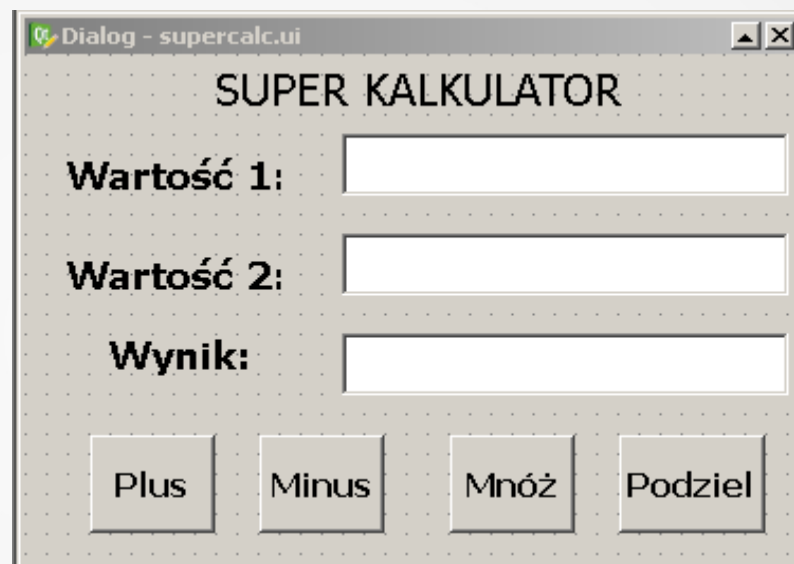
- użycie QTDesigner do stworzenia interfejsu

widżety QT:

**Line Edit**

**Push Button**

**Label**



- implementacja metod dla działań  
**on\_bDodaj\_released(), on\_bMinus\_released()....**
- komentarze do kodu, wysyłanie przez formularz zaliczeń