Wtyczki Python w Quantum GIS.

Wiele programów jest napisanych w taki sposób, aby możliwe było rozszerzenie ich użyteczności, bez ingerencji w kod źródłowy programu. Zazwyczaj odbywa się to poprzez dodanie specyficznie skonstruowanych fragmentów kodu programu, który dodaje pożądane funkcje. Pozwala to osobom trzecim na dodanie funkcjonalności do programu bez potrzeby zapoznawania się z całością, niekiedy bardzo rozległą, konkretnego programu. Utworzone w ten sposób fragmenty kodu działające w połączeniu z innym programem potocznie nazywane są wtyczkami (ang. plugin).

Jednym z programów, który pozwala na taką operację, jest Quantum GIS (QGis) rozpowszechniany na zasadach licencji GPL, i dostępny jest do pobrania ze strony **http://qgis.org**. Pomimo, że licencja nakłada obowiązek rozpowszechniania programu wraz z kodem źródłowym, to modyfikacja tego źródła, napisanego głównie w języku C++, wymaga znacznej wiedzy programistycznej. Dlatego autorzy programu wprowadzili możliwość dodawania nowych elementów w postaci wtyczek napisanych w języku Python.

Python jest stosunkowo nowym językiem programowania, którego pierwsza implementacja powstała na początku lat 90-tych ubiegłego wieku. Określa się go jako język wysokiego poziomu i uniwersalnego zastosowania, który kładzie nacisk na łatwość zrozumienia kodu i możliwość wyrażenia koncepcji w jak najmniejszej jego ilości. Umożliwia programowanie zarówno w sposób strukturalny jak i obiektowy, wraz z możliwością rozdzielenia kodu na mniejsze części, które można zaimportować do programu jako moduły. Pythona uznaje się za język łatwy do nauczenia, jego składnia zapożyczona jest z języka angielskiego, i posiada znacznie mniej wyjątków niż inne, starsze języki. Dodatkowo programy napisane w tym języku bez żadnych zmian mogą być uruchamiane na różnych systemach operacyjnych. Standardowa implementacja Pythona nie posiada funkcji graficznych pozwalających na np. tworzenie okien w nowych systemach operacyjnych, ale jego popularność spowodowała powstanie zewnętrznych bibliotek, które dodają taka funkcjonalność. Jedną z najbardziej popularnych jest biblioteka QT, która używana jest też przez program QGis, przez co jest również oczywistym wyborem dla wtyczek dla tego programu.

Poniższy kurs pomaga w zapoznaniu się z podstawami języka Python, przedstawia jak w stworzyć prosty program w tym języku i jak zastosować tą wiedzę do stworzenia wtyczki w programie Quantum Gis w wersji 2.2.0.

1. Podstawy języka Python

Do zapoznania się z podstawami języka użyta zostanie konsola Pythona, która jest dostępna w programie Quantum Gis. Uruchomić ją można wybierając z menu pozycję ***Wtyczki → Konsola Pythona***. Pozwala na bezpośrednie wydawanie pojedynczych poleceń dla interpretera. W przypadku łączenia wielu poleceń wymagane będzie napisanie skryptu. Użyty w tym celu zostanie program Notepad++, ale można to zrobić w dowolnym innym edytorze tekstu. Specjalny znak „#” oznacza komentarz i tekst za tym znakiem jest ignorowany przez interpreter Pythona. Ponadto należy pamiętać, że każda wartość używana w Pythonie jest obiektem, nawet typ prosty jak ciąg znaków. Co za tym idzie mogą one mieć zdefiniowane metody, czyli funkcje przypisane do tych obiektów.

Dokumentację referencyjna języka można znaleźć tutaj: https://docs.python.org/2/reference/

1.1 Typy danych, zmienne, operatory i struktury

Większość języków programowania wymaga deklaracji typu zmiennej przed jej zastosowaniem. W przypadku Pytona nie jest to niezbędne i deklaracja odbywa się dynamicznie w momencie przypisania wartości. Podstawowymi typami zmiennych są liczby całkowite (ang. integer), zmiennoprzecinkowe (ang. float) i ciągi znaków (ang. string). Przypisanie wartości (obiektu) do zmiennej następuje przez użycie operatora przypisania „=”. Przykładowe przypisania zmiennych można wykonać w konsoli Pythona wpisując poniższe polecenia :

```
a=1          # przypisanie obiektu int „1” do zmiennej a
b="test"     # przypisanie obiektu str do zmiennej b
c=1.0        # przypisanie obiektu float do zmiennej c
```

Następnie można sprawdzić typy zadeklarowanych zmiennych używając funkcji **type()**:

```
type(a)
```

Wyświetlone wartości odpowiadać powinny typom *int, str i float*. Można przy tym zauważyć że typ zmiennoprzecinkowy wymusza się operatorem kropki, nawet jeżeli przypisywana wartość jest liczbą całkowitą. Wartości przypisywane do zmiennych są obiektami, a zmienna jest referencją do tego obiektu. Typy proste są obiektami niemutowanymi, czyli nie można zmieniać ich wartości. Wykonując ponowne przypisanie do zmiennej istniejący obiekt jest usuwany i tworzony jest nowy. Ma to znaczenie przy ciągach w których nie można np. usunąć bezpośrednio jego fragmentu. Żeby uzyskać taki efekt należy utworzyć nowy obiekt składający się z wybranych podciągów. Służy do tego mechanizm cięcia (ang. slicing):

```
'Ala ma kota'[3:8] # wycięcie zakresu znaków od 3 do 8 z obiektu
b[:2]              # wycięcie od początku do pozycji 2
('nasz'+b)[-6:]    # wycięcie poczynając od 6 znaków od końca, do końca
```

Python posiada też specjalny obiekt o nazwie None, którego przypisanie oznacza, że zmienna nie ma wartości, czyli nie wskazuje na żaden obiekt. Zmienne typu ciąg można oznaczać zarówno cudzysłowem jak i apostrofem. Ciągi, jako obiekt, mają też zdefiniowane kilkadziesiąt metod których listę można znaleźć pod adresem https://docs.python.org/release/2.5.2/lib/string-methods.html:

```
b.capitalize()    # Tworzy obiekt typu ciąg z pierwszą dużą literą
'Ala ma kota'.find('kot') # Zwraca pozycję ciągu 'kot'
```

Oprócz typów prostych jak liczby i ciągi dostępne są wbudowane typy złożone mutowane takie jak lista (uporządkowana), zbiór (nieuporządkowany) lub słownik (ang. *list, set, dict*), oraz ich wersje niemutowane czyli tupla (uporządkowana) i zamrożony zbiór (nieuporządkowany) (ang. *tuple, frozenset*).

```
lista=['wpis 0', 'wpis 1', 2, 'wpis 3', 2.0]
zbior=set(('wpis 0', 'wpis 1', 2, 'wpis 3', 2.0))
tupla=('wpis 0', 'wpis 1', 2, 'wpis 3', 2.0)
zbiorz=frozenset(('wpis 0', 'wpis 1', 2, 'wpis 3', 2.0))
slownik={0: 'wpis 0', 1:'wpis 1', 'drugi':2, 'trzeci':'wpis 3', 4:2.0}
```

Do wyświetlenia efektów przypisań można posłużyć się operatorem *print* oraz funkcją type():

```
print lista
print zbior
print tupla[3]
print slownik['trzeci']
type(zbiorz)
```

Po wyświetleniu zauważyć można łatwo różnice pomiędzy strukturami uporządkowanymi i nieuporządkowanymi. W przypadku wersji mutowanych i niemutowanych różnice są mniej oczywiste, ale stają się jasne gdy weźmiemy pod uwagę, że struktura słownika wymaga klucza, który musi mieć wartości unikalne i być obiektem niemutowanym.

Do obiektów i zmiennych na nie wskazujących można zastosować, oprócz przypisania, inne

operatory np. matematyczne:

```
2+5   # dodawanie obiektów typu int
a-3   # odejmowanie zmiennej typu int i obiektu typu int
a*c   # mnożenie zmiennych typu int i float
c/2   # dzielenie zmiennej typu float przez obiekt typu int
```

Bez operatora przypisania interpreter po wpisaniu wyrażenia od razu wyświetli wynik, czyli domyślnie zastosuje operator *print*. Zwrócić należy uwagę, że gdy operacje są wykonywane pomiędzy dwoma liczbowymi typami danych *int* i *float*, to wynik zwracany jest jako typ *float*. Operacje dodawania i mnożenia można wykonywać też na ciągach:

```
”abc”+” aaa”      # dodawanie ciągów
b*3               # mnożenie ciągu
b[:1]+b[2:]       # wycięcie drugiej litery z ciągu
```

Natomiast na strukturach złożonych można dokonywać operacji cięcia, tak jak w przypadku ciągów:

```
tupla[2:3]
lista[3:]
```

Oprócz operatorów matematycznych dostępne są także operatory porównania i logiczne. Operatory porównania pozwalają na sprawdzenie czy i jak obiekty się różnią od siebie:

```
10 == 10
10 != 10
10 >= 3
23 < 3
```

Wynikiem tych działań będzie odpowiedź czy dane wyrażenie jest prawdą czy fałszem, czyli przedstawione jako wartość logiczna 0 (False) lub 1 (True). Wartości logiczne można łączyć używając operatorów logicznych wg tabeli:

And	Or	Not
1 and 1 = 1	1 or 1 = 1	not 1 = 0
1 and 0 = 0	1 or 0 = 1	not 0 = 1
0 and 1 = 0	0 or 1 = 1	
0 and 0 = 0	0 or 0 = 0	

Czyli przykładowe wyrażenie zawierające porównanie i operator logiczny może wyglądać:

```
10 == 10 or 10>=3 # prawda, jeden z warunków jest spełniony
3 < 4 and 4 < 10  # prawda, oba warunki są spełnione
3 < 4 < 10        # uproszczenie zapisu
6 < 4 < 10        # fałsz, jeden z warunków nie jest spełniony
```

Często zdarza się, że podczas wykonania programu potrzebne jest użycie znaków normalnie nie wyświetlanych, lub stosowanych w składni języka. Ma to miejsce najczęściej w przypadku wstawiania znaku końca linii. Aby wykonać taką operacje musimy zastosować jedną z sekwencji ucieczki. Najczęściej stosowanymi są:

```
\n    ASCII linefeed (LF) - nowy wiersz
\r    ASCII Carriage Return (CR) - drugi znak nowego wiersza (Windows)
\'    apostrof
\"    cudzysłów
\\    Backslash (\)
\a    ASCII bell (BEL)
\b    ASCII backspace (BS)
\t ASCII    Horizontal Tab (TAB)
```

Przykładowy ciąg ilustrujący zastosowanie sekwencji może wyglądać następująco:

```
print „ Ala ma kota \n o imieniu \„Mruczek\” ”
```

1.2 Skrypty i kontrola przepływu programu

Połączenie różnych poleceń w jeden program wymaga stworzenia pliku tekstowego zawierającego te polecenia, oraz w zależności od języka jakim się posługujemy, muszą zostać zachowane odpowiednie zasady formatowania. W przypadku Pythona pliki z kodem źródłowym tradycyjnie mają rozszerzenie „.py”, ale odpowiednio sformatowany plik z innym rozszerzeniem również wykona się poprawnie. Takie pliki nazywane są najczęściej skryptami, ponieważ nie jest wymagana ich kompilacja do kodu maszynowego przed wykonaniem, jeżeli komputer na którym są wykonywane ma zainstalowany interpreter danego języka. Podczas pierwszego wykonania skryptu interpreter Pythona tworzy plik z rozszerzeniem „.pyc”, który zawiera skompilowaną wersję skryptu, i jest ona używana dopóki kod źródłowy w skrypcie się nie zmieni. Można wymusić ponowną kompilację usuwając plik „.pyc”.

Składnia programu Python nie wymaga stosowania nawiasów, średników czy tez słów kluczowych aby oznaczyć blok programu. Używane w tym celu jest odpowiednio dopasowane wcięcie, czyli dodanie spacji na początku linii. Czyli poprawnie napisany program jednocześnie jest dobrze sformatowany pod względem estetycznym. Plik skryptu można stworzyć używając dowolnego edytora, na przykład standardowego notatnika. Jednak, żeby uzyskać podświetlanie składni dobrze jest użyć innego programu, takiego jak Notepad++. Pierwszy prosty program możemy napisać łącząc następujące wyrażenia:

```
a=9
b=3
wyn=9+3
print 'Wynik dodawania ',a,' i ', b, ' : ', wyn
```

Plik zapisać możemy w katalogu d:\temp\python, nazywając go „test.py”. Aby uruchomić program należy uruchomić konsolę OS4Geo4W znajdującą się tam gdzie skrót do QGis. Następnie należy zmienić katalog na ten, w którym znajduje się plik skryptu i uruchomić interpreter z nazwą skryptu jako argument:

```
d:
cd d:\temp\python
python test.py
```

Przed rozpoczęciem dalszej zmiany programu warto dodać, że zmienne w Pytonie w zasadzie mogą mieć dowolne nazwy, jeżeli nie kolidują z zarezerwowanymi słowami kluczowymi, mianowicie:

```
and as assert break class continue def del elif else except exec finally for
from global if import in is lambda not or pass print raise return try while
with yield
```

Oprócz tych słów kluczowych standardowa implementacja zawiera oczywiście wiele funkcji i metod, ale maja one nawias po nazwie, więc nie zachodzi obawa konfliktu.

Co prawda polecenia z pierwszego skryptu z powodzeniem można wpisać ręcznie w konsoli Pythona, ale często musimy wykonać wiele podobnych operacji, gdzie ręczne wpisywanie było by zbyt uciążliwe. Z pomocą przychodzi konstrukcja pętli, gdzie można zastosować jedną z dwóch: *while* lub *for*. Konstrukcja *while* wykonuje się tak długo, jak długo warunek ją kontrolujący jest *Prawdą*:

```
print 'Zgadnij cyfre od 1 do 9'
while input() != 8:             # czy wpisana cyfra się zgadza?
      print 'Zla odpowiedz!'    # jeżeli nie
print 'Trafiles'                # jeżeli tak
```

Nowa funkcja *input*() czyta wpisaną wartość z klawiatury, ewaluuje ją, a następnie sprawdzane jest czy wartość ta różni się od ośmiu. Pętla będzie wykonywać się tak długo jak długo ten warunek jest prawdą. Zwrócić uwagę należy na wspomniane wcześniej wcięcie, które w tym przypadku kontroluje która część kodu będzie wykonywać się w pętli.

Druga wersja pętli w programie ject konstrukcja *for* i w przeciwieństwie do *while* nie korzysta z warunku,

tylko z zdefiniowanego zakresu wartości:

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

Nowa funkcja range() tworzy listę wartości liczbowych o określonym zakresie. Należy zwrócić uwagę na zakończenie operatora print przecinkiem. Oznacza to, że nie chcemy aby wyświetlony został koniec linii. Można też zauważyć, że w argumencie operatora print pojawił się znak „\n”. Jest to tak zwany znak ucieczki (ang. *escape sequence*) i pozwala na wyświetlanie znaków których nie ma na klawiaturze, np. koniec linii, tak jak to ma miejsce w tym przypadku. Nowy operator *in* sprawdza czy wartość znajduje się w zbiorze.

W praktycznie każdym programie występuje miejsce, w którym w zależności od jakiejś wartości chcemy aby program wykonał inną operację. Miało to miejsce co prawda w przypadku pętli while, ale nie zawsze pętla jest potrzebna. W tej sytuacji będziemy korzystać z konstrukcji ***if..else****:*

```
print 'Czy chcesz kontynuowac (t/n)?'
k=raw_input()
if k == 't':
      print 'Kontynuuje program'
elif k == 'n':
      print 'Zatrzymuje program'
else:
      print 'Zla opcja'
```

Zastosowana funkcja raw_input() ma podobne działanie jak input(), ale traktuje wpisane wartości jako ciąg bez ewaluowania wartości.

Rozbudowując program możemy stwierdzić, że niektóre fragmenty są za długie, i chcielibyśmy rozdzielić je na mniejsze części. Użyjemy wtedy konstrukcji funkcji, która jest osobnym blokiem i można ją wywołać z dowolnego miejsca i w razie potrzeby z parametrami:

```
def dodaj(a,b):   # definicja funkcji
      c=a+b
      return c    # zwraca wartośc do programu
print dodaj(2,3)  # początek programu
wyn=dodaj         # funkcje można przypisać do zmiennej
print wyn(3,4)
```

1.3 Klasy i moduły

Podążając trendem do rozbudowy programu może okazać się, że sama możliwość podzielenia programu na funkcje to za mało, i potrzebny będzie kolejny poziom abstrakcji, żeby je dodatkowo usystematyzować. W tej sytuacji nie pozostaje nic innego jak podzielenie funkcji na grupy nazywane *klasami.* Jednocześnie funkcje należące do danej klasy zaczniemy nazywać *metodami*. Ponadto sposób w jaki programujemy zmieni się ze strukturalnego na obiektowy. Oczywiście klasa to struktura potencjalnie o wiele bardziej złożona niż po prostu zagnieżdżenie funkcji, bo może ona dziedziczyć metody z klasy bazowej lub przechowywać dowolne inne typy danych. Przykładową klasę możemy zdefiniować w następujący sposób:

```
class Kalk(object):
      def __init__(self,s='Inicjalizuje klase kalkulatora'):
            self.zaps = s     # przypisanie zmiennej do zmiennej lokalnej
            self.hist=[]      # inicjalizacja pustej listy
            print self.zaps   # wyswietlenie napisu

nowykalk1=Kalk()                              # początek programu
nowykalk2=Kalk('Tworze klase z parametrem')   # drugi obiekt
```

Powyższa klasa na razie jeszcze nic nie robi poza swoja inicjalizacją, co powoduje stworzenie dwóch obiektów na które wskazują zmienne nowykalk1 i nowykalk2. Jedna metoda która się w niej znajduje to wbudowana funkcja inicjalizacji która wykonuje się raz przy tworzeniu obiektu. W jej parametrach widzimy wyrażenie „self", które jest automatycznie tworzonym odniesieniem do aktualnie tworzonego obiektu. Drugim parametrem jest „s" przy którym widzimy, że ma przypisaną wartość domyślną, a co za tym idzie jest opcjonalny. Nowym elementem jest również desygnator w postaci kropki który pozwala na odwoływanie się do metod lub zmiennych lokalnych klasy. Słowo „object" oznacza że klasa Kalk dziedziczy z klasy nadrzędnej (bazowej,super-klasy) o tej nazwie. Spróbujmy więc rozbudować klasę Kalkulatora o metody, które pozwolą na wykonywanie działań:

```
class Kalk(object):
      (...)
      def dodaj(self,a,b):
            self.hist.append(a+b)
            return a+b
      def mnoz(self,a,b):
            self.hist.append(a*b)
            return a*b
      (...)
      def histo(self,res=0):
            if res:
                  self.hist=[]
            else:
                  print self.hist[:]

(...)                         #początek programu
print nowykalk1.dodaj(3,4)
print nowykalk1.mnoz(3,4)
(...)
nowykalk1.histo()
```

Dodatkowo oprócz działań w przykładzie mamy metodę histo która pozwala na wyświetlenie historii wyników, które przechowywane są jako wpisy listy. A wysłanie parametru 1 lub True do tej metody spowoduje jej wyczyszczenie.

Ostatnim stopniem kompartmentacji programu jest jego rozdzielenie na osobne pliki, które nazywane są modułami. Klasy i funkcje z tych modułów mogą zostać udostępnione w aktualnym programie poleceniem *import.* Korzystając z tej funkcjonalności możemy skorzystać z naszej klasy zapisanej w pliku test.py, tworząc nowy plik *test2.py,* w którym zaimportujemy moduł *test* rozbudować program o interaktywny interfejs korzystając z pętli *while. Należy pamiętać o usunięciu części programu i pozostawieniu tylko klasy w pliku test.py*:

```
import test

nowykalk=test.Kalk()    # tworzenie obiektu z klasy w module „test"
d=None
while d!='x':
      print 'Podaj dzialanie (+,-,*,/,h), lub x aby zakonczyc'
      d=raw_input()
      if d in ['+','-','*']:
            print 'Podaj pierwsza liczbe:'
            a=input()
            print 'Podaj druga liczbe:'
            b=input()
            if d=='+':
                  print 'Wynik: ',nowykalk.dodaj(a,b)
            elif d=='*':
                  print 'Wynik: ',nowykalk.mnoz(a,b)
            (...)
      elif d=='h':
            nowykalk.histo()
```

```
      elif d!='x':
            print 'Nieznana opcja'
```

Po tej operacji uruchamiamy program poleceniem:

```
      python test2.py
```

Python posiada szeroką gamę standardowych bibliotek, oraz jeszcze większą takich które zostały stworzone niezależnie od jego twórców. Wszystkie możemy zaimportować tak samo jak nasz plik:

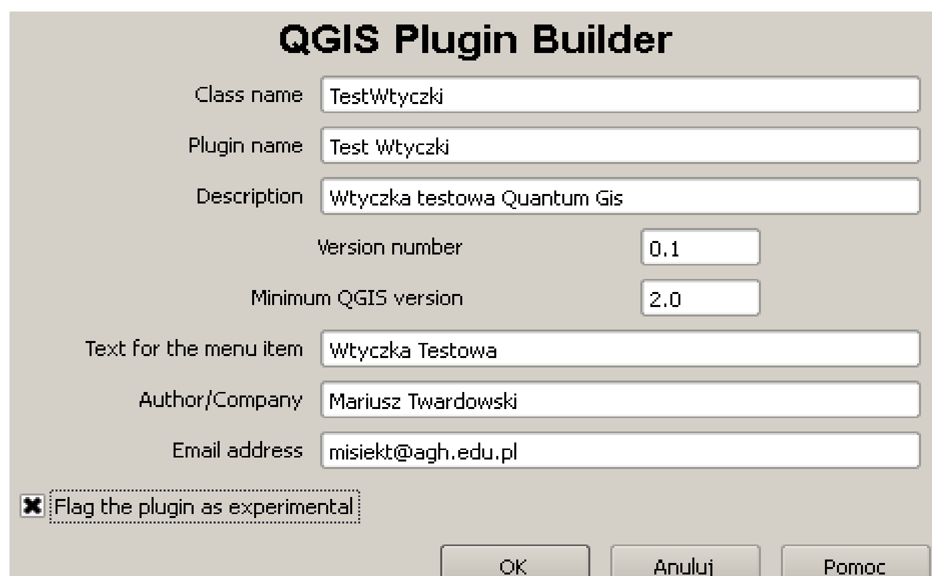
```
import math, sys     # zaimportuj moduł funkcji matematycznych i systemowych
from PyQt4 import *  # wszystkie klasy z biblioteki graficznej PyQt4
dir(math)            # sprawdź zaimportowane funkcje z modułu math
```

2. Budowa wtyczki QGis

Wprowadzenie z pierwszej części nie wyczerpuje oczywiście wszystkich aspektów języka Python, jednak pozwoli na łatwiejsze zrozumienie dalszej części, w której opisana będzie budowa właściwego programu czyli wtyczki QGis. W tej części będzie można zapoznać się z podstawowym sposobem wygenerowania odpowiedniego kodu, który doda odpowiedni wpis w menu programu i zainicjalizuje podstawowe okno. Następnie po krótkiej analizie tego kodu, korzystając z wiedzy nabytej w części pierwszej dodane zostaną elementy odpowiedzialne za wymaganą funkcjonalność wtyczki.

2.1 Generowanie wtyczki ze schematu

Korzystając z faktu, że istnieje już wtyczka która generuje podstawowy schemat kodu działającego w Quantum Gis, możemy jej użyć aby znacznie przyspieszyć naszą pracę. Po uruchomieniu QGis 2.2.0, w menu *Wtyczki->Zarządzaj wtyczkami* wybieramy *Ustawienia* i zaznaczamy *Wtyczki eksperymentalne*. Następnie znajdujemy wtyczki **Plugin Builder** oraz **Plugin Reloader** i jeżeli nie są zainstalowane to wybieramy przycisk *Instaluj Wtyczkę*. Po tej operacji wtyczki powinny pojawić się w tym samym menu programu. Następnie uruchamiamy wtyczkę Plugin Builder i wypełniamy wymagane pola. Pola opcjonalne mogą pozostać puste.



Po wypełnieniu odpowiednich pól i kliknięciu OK wybieramy katalog w którym schemat wtyczki ma być zapisany. Może być to dowolna lokalizacja, ale najlepiej skorzystać od razu z lokalizacji przeznaczonej dla wtyczek:

```
C:\Users\Student\.qgis2\python\plugins
```

Następnie uruchamiamy konsolę OSGeo4W, zmieniamy katalog na wymieniony powyżej, ale wraz z nazwą wtyczki, i generujemy kod Pythona z plików interfejsu i ikony, czyli wykonujemy polecenia:

```
c:
cd C:\Users\Student\.qgis2\python\plugins\TestWtyczki
pyuic4 -o ui_testwtyczki.py ui_testwtyczki.ui
pyrcc4 -o resources_rc.py resources.qrc
```

Po tej operacji restartujemy Quantum Gis, wchodzimy do zarządzania wtyczkami, znajdujemy naszą wtyczkę testową na liście i ją włączamy. W menu wtyczek powinna pojawić się zdefiniowana przez nas wtyczka. W tym momencie można byłoby stwierdzić, że temat został wyczerpany, ponieważ uzyskaliśmy zakładany efekt. Jednak od razu widać, że chociaż wtyczka działa, to nie ma żadnej funkcjonalności, i ma tylko przyciski OK i Anuluj, które również nic nie robią. Tutaj właśnie zaczniemy wykorzystywać informacje z poprzedniej części kursu.

2.2 Zawartość wygenerowanej wtyczki

Analizując pliki wygenerowane przez wtyczkę Plugin Builder możemy określić w jaki sposób aplikacja działa. W pierwszej kolejności zobaczymy, że wygenerowane nazwy plików zostały zapożyczone z nazwy wtyczki którą zdefiniowaliśmy w generatorze. Następnie sprawdzając zawartość pliku **__init__.py** możemy stwierdzić, że zawiera inicjalizację naszego testowego pluginu, umożliwiając programowi QGis jego zarejestrowanie w interfejsie. Wewnątrz nie ma wiele, tylko jedna funkcja, ale istotnym elementem jest import klasy *TestWtyczki* z modułu **testwtyczki**. Korzystając z poprzednich informacji wiemy, że oznacza to, że klasa zostanie wczytana z pliku **testwtyczki.py**

Przechodząc do wskazanego przez inicjalizację pliku możemy zauważyć, że jest on o wiele bardziej rozbudowany niż poprzedni. Ale większość z wpisów w tym pliku to dalsze definicje pewnych zmiennych i translacji wymaganych dla QGis. Głównymi interesującymi dla nas wpisami są:

```
import resources_rc
from testwtyczkidialog import TestWtyczkiDialog
      class Testwtyczki:
            (…)
            self.dlg=TestWtyczkiDialog()
```

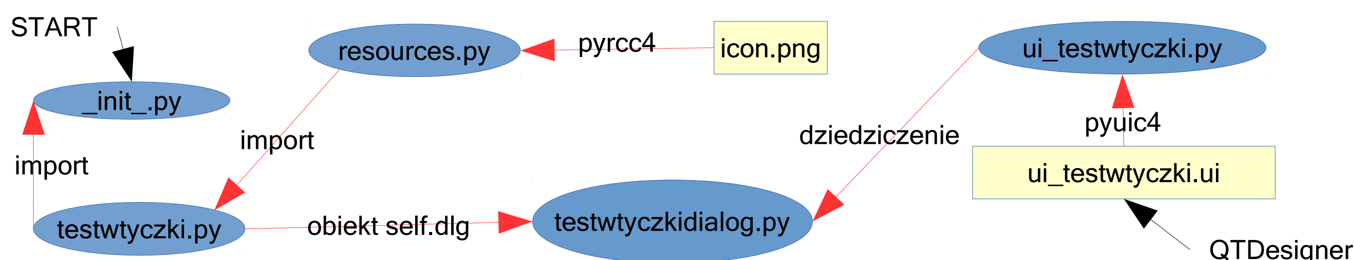
Pierwszy wpis importuje moduł **resources_rc.py,** który zawiera w zmiennej *qt_resource_data* reprezentację szesnastkową ikony zawartej w pliku **icon.png,** i został wygenerowany poprzednio z pliku xml **resources.qrc** przy pomocy narzędzia **pyrcc4.** Można oczywiście narysować własną wersje ikony dla wtyczki, i ponownie wygenerować jej kod. Drugi wpis jest bardziej istotny, i informuje nas o tym, że z pliku **testwtyczkidialog.py** zaimportowana zostanie klasa *TestWtyczkiDialog*. Przy jej użyciu trochę poniżej stworzony zostaje obiekt o nazwie **self.dlg**.

Kontynuując analizę przepływu programu sprawdzamy zaimportowany plik **testwtyczkidialog.py**, i możemy od razu rozpoznać że powoduje on głównie dwie akcje:

```
from ui_testwtyczki import Ui_TestWtyczki
class TestWtyczkiDialog(QtGui.QDialog, Ui_TestWtyczki):
      (…)
      self.setupUi(self)
```

Pierwsza akcja to import klasy **Ui_TestWyczki** z pliku **ui_testwtyczki.py**, natomiast druga to wywołanie metody **self.setupUi()**, która powoduje ustawienie różnych parametrów interfejsu wtyczki. Zwrócić należy uwagę, że w tym przypadku nie jest tworzony nowy obiekt, ale klasa **Ui_TestWtyczki** jest *dziedziczona* przez klasę z aktualnego pliku, czyli **TestWtyczkiDialog**. Co za tym idzie wszystkie metody klasy bazowej są dla niej dostępne, łącznie z metodą **setupUi()**.

Ostatecznie przechodzimy do pliku **ui_testwtyczki.py**, i widzimy że jest on dość skomplikowany. Na szczęście nie będziemy go musieli edytować, ponieważ jest on generowany z pliku **ui_testwtyczki.ui**, o którym później. W efekcie możemy narysować następujący diagram naszej aplikacji:



Rysunek 1: Schemat programu

2.3 Implementacja programu

Po zapoznaniu się z podstawową budową wtyczki QGis należy zdecydować co ma być celem działania naszego programu. Na potrzeby tego kursu wybrano przeliczenie współrzędnych z Krakowskiego Układu Lokalnego (**KUL**) do wymaganego przez UMK układu 2000, a dla Krakowa strefy 7 czyli układu o kodzie EPSG:2178. Przeliczenia dokonuje się za pomocą transformacji konforemnej 2-go stopnia, opracowanej przez Piotra Banasika, która ma postać:

```
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)
```

gdzie współczynniki wynoszą:

```
xbp=-3.04419376190476×10^4    ybp= 2.89030878571429×10^5
xbw= 5.54659313500000×106     ybw= 7.42852359576190×106
x=x-xbp                       y=y-ybp
a1 = 2.73660450713879×10^-22  a2 = 7.69215691432381×10^-2
a3 = -9.99116225304304×10^-1  a4 = 4.05478646361601×10^-2
a5 = 6.77548043232547×10^-11  a6 = 9.08724778098026×10^-10
```

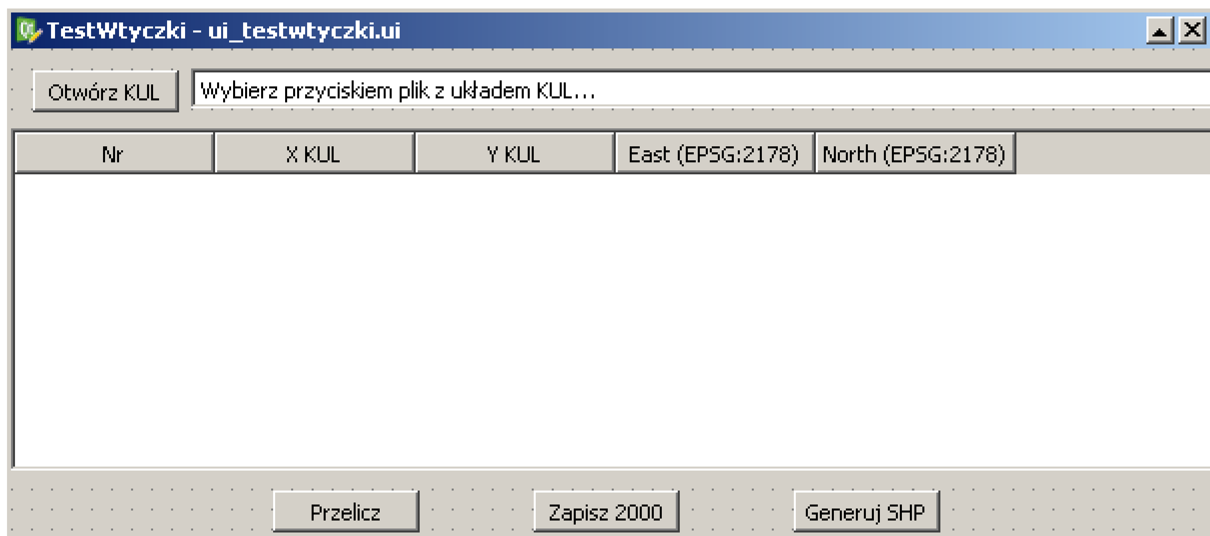
Wejściowymi danymi będzie plik tekstowy ze współrzędnymi punktów w KUL. Program powinien wczytać numery i współrzędne punktów do tabeli, przeliczyć je na układ 2000, i ostatecznie powinien wygenerować plik tekstowy z nowymi wartościami.

2.3.1 Interfejs programu

Pierwszym elementem który musimy zaprojektować jest interfejs użytkownika, który pozwoli na interakcje z programem. Klasa zawierająca ten interfejs została już wspomniana, i znajduje się w pliku **ui_testwtyczki.py**. Jednak nie będziemy tego pliku modyfikować ręcznie, tylko wykorzystamy pokazane na rysunku 1 narzędzie **QTDesigner**. Pozwoli ono na zaprojektowanie wszystkich elementów i zapisanie ich do pliku **ui_testwtyczki.ui**, z którego za pomocą narzędzia **pyuic4** wygenerowany zostanie kod Pythona. Program QTDesigner uruchamiamy z lokalizacji:

```
c:\Program Files (x86)\QGIS Valmiera\bin\designer.exe
```

Po uruchomieniu programu otwieramy istniejący plik interfejsu **ui_testwtyczki.ui** znajdujący się w katalogu z wtyczką. Pojawi się podstawowy dialog wygenerowany przez Plugin Buildera, a naszym zadaniem będzie jego modyfikacja. Docelowo będziemy chcieli, żeby dialog wyglądał następująco:



Rozpoczynamy od zaznaczenia i usunięcia domyślnych przycisków OK i Anuluj. Następnie klikamy na pusty dialog i w *Edytorze właściwości* zmieniamy geometrię dialogu z 400x300 na 600x400. Następnie dodajemy do dialogu poszczególne elementy przeciągając je z listy po lewej stronie:

- przycisk „Otwórz KUL”: przeciągamy element „**Push Button**”, a następnie prawym przyciskiem wybieramy ***Zmień tekst...*** i zmieniamy widoczny tekst na „Otwórz ULK”. Oprócz tego prawym przyciskiem wybieramy opcję ***Zmień nazwę obiektu...*** i zamieniamy ją na **bOpenKUL**,
- linię „Wybierz przyciskiem....”: przeciągamy element „**Line Edit**” zmieniamy tekst na taki jaki jest widoczny na obrazku, a następnie zmieniamy nazwę obiektu na **pathKUL**. W Edytorze właściwości zaznaczamy flagę **readOnly**,
- tabelę główną: przeciągamy element „**Table Widget**”, dopasowujemy jego rozmiar do okienka i zmieniamy nazwę obiektu na **coordTable**. Dodatkowo prawym przyciskiem wybieramy ***Modyfikuj elementy...*** i dodajemy kolumny Nr, X_KUL, Y_KUL, East (EPSG:2178), North (EPSG:2178),
- przycisk „Przelicz”: „Push Button” zmieniamy tekst, a nazwę obiektu na **bCalc**,
- przycisk „Zapisz 2000”: „**Push Button**”, zmieniamy tekst a potem nazwę obiektu na **bSave2000**,
- przycisk „Generuj SHP”: **Push Button**”, zmieniamy tekst a potem nazwę obiektu na **bGenSHP**

Plik interfejsu zapisujemy i używając konsoli GDAL generujemy plik **ui_testwtyczki.py** poleceniem **pyuic4** tak samo jak w punkcie 2.1. Uruchamiamy w QGis wtyczkę aby sprawdzić czy interfejs poprawnie się wyświetla. Możliwe że będzie konieczne użycie **Plugin Reloadera**.

2.3.2 Implementacja funkcji

Następnym etapem będzie implementacja kodu Pythona, który pozwoli na wykonywanie odpowiednich operacji w zależności od wydanych poleceń. Biblioteka **PyQt,** której użyliśmy do stworzenia interfejsu, posiada specjalny mechanizm, który bardzo ułatwia łączenie interfejsu z implementacją i kryje się pod nazwą **sygnałów** i **slotów.** Sygnał to wywołanie generowane przez np. naciśnięty przycisk interfejsu, który może wysłać domyślne sygnały clicked(), pushed(), released(). Natomiast **slot** to nic innego jak metoda klasy, która można połączyć z sygnałem funkcją **connect()** np.:

```
self.bOpenKUL.clicked.connect(self.funkcjaOtwieraniaPliku)
```

Powyższa metoda połączyłaby przycisk „Otwórz KUL” z metodą „funkcjaOtwieraniaPliku”, czyli po naciśnięciu przycisku metoda była by wykonana. Co więcej, PyQt automatycznie tworzy połączenia do metod które nazwane są zgodnie ze schematem „**on_nazwaObiektu_sygnał()**”, czyli powyższe można pominąć stosując nazwę metody **on_bOpenKUL_connect().** Z tej funkcjonalności będziemy korzystać, ponieważ kod interfejsu wygenerowany z QTDesignera zawiera metodę **connectSlotsByName()**, która to umożliwia.

Metody do obsługi interfejsu będziemy dodawać w pliku **testwtyczkidialog.py**, ale zaczniemy od dodania modułów na początku i zadeklarowania kilku zmiennych w metodzie **__init__**:

```
import os, qgis
from osgeo import osr, ogr
(...)
    def __init__(self):
       (...)
         self.tabKUL=[]  # lista/tablica na numery i współrzędne KUL
         self.tab2K=[]   # tablica na współrzędne przeliczone 2000/7
         self.iK={'nr': 0, 'x':1, 'y':2, 'E':3, 'N':4} # indeks kolumny
```

Dwie pierwsze deklaracje nie wymagają wyjaśnienia, natomiast trzecia umożliwi nam odwoływanie się do numeru kolumny tabeli przy pomocy liter zamiast cyfr, co pozwoli na lepszą przejrzystość kodu programu.

W drugim kroku dodamy metodę dzięki której naciśnięcie przycisku „Otwórz KUL” pozwoli na wczytanie pliku i dodanie wartości do tabeli:

```
def on_bOpenKUL_released(self):
    opfi=QtGui.QFileDialog.getOpenFileName  # przypisanie metody do zmiennej
    nazwaKUL=opfi(self, 'Wybierz KUL','', 'TXT files (*.txt);;ALL Files (*.*)')
    self.pathKUL.setText(nazwaKUL)          # ustawienie tekstu w linii
    with open(str(nazwaKUL),'r') as plik:   # otwarcie pliku
        tab=self.crdTable                 # przypisanie zmiennej lokalnej/ skrót
        wI=QtGui.QTableWidgetItem           # -||-
        for linia in plik:                  # pętla dla lini w pliku
            nr, x, y = linia.split()      # przydzielenie wartości do zmiennych
            self.tabKUL.append([int(nr), float(x), float(y)]) # wart. do tab.
            tab.insertRow(tab.rowCount())  # dodaj wiersz w tabeli
            tab.setItem(tab.rowCount()-1, self.iK['nr'], wI(nr)) #wart kom. 0
            tab.setItem(tab.rowCount()-1, self.iK['x'], wI(x)) #wart kom. 1
            tab.setItem(tab.rowCount()-1, self.iK['y'], wI(y)) #wart kom. 2
```

Widzimy automatyczne połączenie obiektu **bOpenKUL** z sygnałem **released()**, przypisujemy też długie nazwy funkcji do krótkich zmiennych lokalnych **opfi**, **tab**, **wI**, co nie jest niezbędne ale zwiększa czytelność. Stosujemy strukturę **with...as...else,** która pozwala na otwarcie pliku, lub pominięcie kodu zawartego w jej suicie, jeżeli otwarcie pliku się nie powiedzie. Nie mamy jednak w tej konstrukcji możliwości powiadomienia o tym fakcie użytkownika, ponieważ trzeba wtedy zastosować obsługę wyjątków, ale o tym później. Ostatecznie wykonywana jest pętla, w której wstawiane są wartości do tablicy **self.tabKUL** oraz wypełniana jest tabela widgetu w obiekcie **self.crdTable**. Teraz możemy przetestować wczytywanie współrzędnych do tabeli.

Następnie dodamy prostą metodę do wykonania transformacji konforemnej. Wartości współczynników można skopiować z http://fotogrametria.agh.edu.pl/~misiek/python/wspolczynniki.txt

```
def calcK2k(self, x, y):
    xbp, ybp = -3.04419376190476*(10**4),   2.89030878571429*(10**5)
    xbw, ybw =  5.54659313500000*(10**6),   7.42852359576190*(10**6)
    a1,  a2  =  2.73660450713879*(10**-2),  7.69215691432381*(10**-2)
    a3,  a4  = -9.99116225304304*(10**-1),  4.05478646361601*(10**-2)
    a5,  a6  =  6.77548043232547*(10**-11), 9.08724778098026*(10**-10)
    x,y = x-xbp, y-ybp
    N = xbw + a1 + a3*x - a4*y + a5*(x*x - y*y) - 2*a6*x*y
    E = ybw + a2 + a3*y + a4*x + 2*a5*x*y + a6*(x*x - y*y)
    return E, N
```

Nie ma w niej nic nadzwyczajnego, podstawiane są wartości do zmiennych współczynników, a potem wykonywane są obliczenia zmiennych E i N. Są to współrzędne w układzie 2000/7, które są zwracane przez metodę.

Gdy możemy już obliczyć nowe współrzędne przystępujemy do zaimplementowania obsługi przycisku „Przelicz”:

```
def on_bCalc_released(self):
    wiersz=0
    wI=QtGui.QTableWidgetItem
    for nr, x, y in self.tabKUL:
        E,N=self.calcK2k(x,y)
        self.tab2K.append([nr,E,N])
        self.crdTable.setItem(wiersz, self.iK['E'], wI(str(E)))
        self.crdTable.setItem(wiersz, self.iK['N'], wI(str(N)))
        wiersz+=1
```

Ponownie widzimy automatyczne połączenie sygnału, tym razem dla obiektu **bCalc**, a następnie pętlę w której odwołujemy się do napisanej przed chwilą metody **calcK2k()**. Zwrócone z metody wartości wpisywane są do tabeli self.tab2K, oraz do 4 i5 kolumny widgetu **self.crdTable**. Po tych operacjach możemy przetestować przeliczanie wartości pomiędzy układami KUL i 2000.

Jeżeli mamy już przeliczone współrzędne, to nie pozostaje nic innego jak zapisać ich wartości do nowego pliku tekstowego:

```
def on_bSave2000_released(self):
    safi=QtGui.QFileDialog.getSaveFileName
    nazwa2k=safi(self, 'Podaj plik 2000','', 'TXT files (*.txt);;ALL (*.*)')
    try:
        plik2k=open(nazwa2k,'w')
    except IOError:
        QtGui.QMessageBox.critical(self,u'Błąd',u'Błąd zapisu pliku')
    else:
        for nr, e, n in self.tab2K:
            plik2k.write('{nr} {e} {n}\n'.format(**locals()))
```

Zauważymy, że są w niej nowe elementy, z których najistotniejszym jest obsługa wyjątków przy zastosowaniu struktury **try...except...else**. Czyli program próbuje otworzyć plik, a jeżeli się to nie powiedzie wyświetlony zostanie komunikat przy użyciu metody **QmessageBox.critical()**. Zastosowano tutaj wyjątek **IOError**, który dotyczy ogólnie wejścia/wyjścia, ale wyjątków jest znacznie więcej. Natomiast jeżeli plik można zapisać, to wykonany zostanie kod w bloku **else**. Nowym elementem jest również zastosowanie znaku **„u"** przed ciągiem, który po prostu pozwala formatować napisy z polskimi znakami. Bez niej wyświetliły by się krzaczki w komunikacie. Ostatnim nowym elementem jest sposób formatowania wartości przy zapisie do pliku w metodzie **plik2k.write()**. Jako argument tej metody wysyłamy obiekt tekstowy, jednocześnie korzystając z jego metody **format()**, ale zamiast definiować tradycyjnie listę zmiennych stosujemy funkcję **locals(),** która automatycznie zwraca wszystkie zmienne lokalne, w naszym przypadku **nr**, **e** i **n**.

Ostatnim etapem będzie zapisanie punktów do pliku SHP i dodanie go jako warstwy do QGIS. Dlatego implementujemy funkcje do obsługi przycisku „Generuj SHP":

```
def on_bGenSHP_released(self):
    shpdrv=ogr.GetDriverByName("ESRI Shapefile")
    safi=QtGui.QFileDialog.getSaveFileName
    scieSHP=safi(self, 'Podaj plik shp','', 'SHP files (*.shp)')
    nazwaWar=os.path.basename(str(scieSHP))[:-4] # nazwa warstwy z pliku
    if os.path.exists(str(scieSHP)):
        shpdrv.DeleteDataSource(str(scieSHP))
    shp=shpdrv.CreateDataSource(str(scieSHP))  # obiekt shape
    srs=osr.SpatialReference()  # obiekt referencji przestrzennej
    srs.ImportFromEPSG(2178)    #utwórz układ 2000/7
    warstwa=shp.CreateLayer(nazwaWar, srs, ogr.wkbPoint) # obiekt warstwy
    pole=ogr.FieldDefn("Nr", ogr.OFTString)
    pole.SetWidth(10)
    warstwa.CreateField(pole)
    warstwa.CreateField(ogr.FieldDefn("East", ogr.OFTReal))
    warstwa.CreateField(ogr.FieldDefn("North", ogr.OFTReal))
    for nr, e, n in self.tab2K:
        punkt=ogr.Feature(warstwa.GetLayerDefn()) # stworzenie obiektu
        punkt.SetField("Nr", str(nr))   # dodaj wartość atrybutu
        punkt.SetField("East", e)
        punkt.SetField("North",n)
        geom=ogr.Geometry(ogr.wkbPoint) # obiekt geometrii
        geom.AddPoint(e, n)
        punkt.SetGeometry(geom) # dodanie geometrii
        warstwa.CreateFeature(punkt) # dodanie punktu do warstwy
    qgis.utils.iface.addVectorLayer(scieSHP,nazwaWar,'ogr')
```

Warstwa powinna pojawić się na liście z lewej strony, musimy jeszcze wybrać prawym przyciskiem na nazwie warstwy opcję „Powiększ do zasięgu warstwy", żeby wyświetlić punkty. Natomiast opcją „Tabela atrybutów" możemy sprawdzić dodane pola dla punktów.

Aby sprawdzić poprawność zaimportowanych danych można sprawdzić współrzędne w układzie geograficznym WGS84, lub pobrać mapę granic z zewnętrznego źródła. Na przykład można posłużyć się serwerem WFS pod adresem http://fotogrametria.agh.edu.pl/geoserver/SIT/wfs i dodać warstwę „World Countries".