Materiały szkoleniowe

Python w QGIS - przetwarzanie i analiza danych (poziom zaawansowany)

Spis treści:

Informacje wstępne 4

Omówienie budowy aplikacji QGIS 5
- Struktura wewnętrzna 5
- Konsola Python w QGIS 6
- Konsola OSGeo4W Shell 7

QGIS API 9
- Biblioteki QGIS 11
- Główne klasy QGIS API 11
- Obiekt iface 12
- Obsługa warstw przestrzennych 13
  - Dane rastrowe 15
  - Dane wektorowe 16
  - Obiekty przestrzenne 17
    - Modyfikacja istniejącej warstwy 18

GIT i systemy kontroli wersji 22

Wtyczki QGIS 23
- Plugin Builder 23
- Struktura wtyczki 28
  - analiza_zasiegu_dockwidget_base.ui 28
  - __init__.py 28
  - analiza_zasiegu.py 28
- Plugin Reloader 30

Framework Qt 32
- Zasoby 32
- Widżety 33
- Qt Designer 35
  - Elementy interfejsu aplikacji Qt Designer 36
  - Układy (layouts) 37
  - Pliki .ui i Python 37
- Sygnały i sloty 43

Rozwijanie wtyczki 47
- Dostęp do widżetów 47
  - QPushButton 48
  - QCheckBox 48
  - QSpinBox i QDoubleSpinBox 48
  - QgsMapLayerComboBox 48
  - QgsFieldComboBox 48

Akcje 50
Narzędzia mapy 53
QgsMapToolEmitPoint 53
Aktywacja narzędzi mapy 54
Akcje i narzędzia mapy 54
Rysowanie w oknie mapy 59
Klasa QgsRubberBand 59
Kształt 59
Stylizacja 60
Kolory 60
Analizy przestrzenne i automatyzacja procesów przetwarzania danych 64
Informacje o algorytmach 64
Wywoływanie algorytmów z poziomu języka Python 66

Informacje wstępne

Szkolenie ma na celu wprowadzenie do tworzenia wtyczek do aplikacji QGIS w języku Python, ze szczególnym uwzględnieniem analiz przestrzennych. Po jego zakończeniu uczestnicy będą znać zasady działania rozszerzeń w tym środowisku, poruszać się w dokumentacji QGIS API w celu wyszukania potrzebnych informacji,
Szkolenie ma poziom zaawansowany i wymagana jest od uczestników przynajmniej podstawowa znajomość języka programowania Python oraz programowania obiektowego. Ćwiczenia, szczególnie dotyczące tworzenia wtyczki, są ze sobą powiązane i należy je wykonywać zgodnie z ich kolejnością w niniejszym dokumencie.
Wszystkie ćwiczenia i materiały zostały stworzone z pomocą QGIS w wersji 3.16, która jest wersją o wydłużonym wsparciu (*LTR - Long Term Release*) wg stanu na czas ich tworzenia (sierpień 2021 r.). Możliwe jest również wykorzystanie innej wersji QGIS, jednak w zależności od wprowadzonych w niej zmian mogą być konieczne zmiany w kodzie źródłowym wtyczki ćwiczeniowej, które je uwzględnią.
Dodatkowo, w związku z koniecznością pisania kodu źródłowego w języku Pythona uczestnicy powinni zainstalować edytor tekstowy umożliwiający kolorowanie składni. W trakcie ćwiczeń wykorzystywany będzie program *Notepad++*, ale uczestnicy mogą korzystać dowolną aplikację posiadającą tą funkcję np. *Visual Studio Code*, *PyCharm*, *Atom*, *Sublime Text* itp.
Do stworzenia części materiałów wykorzystano informacje ze szkolenia *Python w QGIS - poziom średniozaawansowany*, które było realizowane w 2020 r. dla Ministerstwa Klimatu (http://ekoportal.gov.pl/fileadmin/user_upload/Python_w_QGIS_materialy_szkoleniowe.pdf) .
Materiały z tego i innych szkoleń są do pobrania ze strony http://ekoportal.gov.pl/ .
Materiały szkoleniowe zawierają dodatkowe pliki pogrupowane w podkatalogi.

analiza_zasiegu
Zawiera kod wtyczki, która będzie wykonywana w ramach ćwiczeń. Jest on wersjonowany za pomocą systemu GIT, dzięki czemu możliwe jest włączenie wersji kodu źródłowego odpowiadającego konkretnemu ćwiczeniu w niniejszym skrypcie. Narzędzie GIT jest dołączone do materiałów i nie ma potrzeby jego osobnej instalacji.

projekt
Projekt z danymi przestrzennymi, które będą wykorzystywane w ramach ćwiczeń. Należy go wczytać do QGIS przed ich rozpoczęciem.

pliki_wtyczki
Są to pliki ikon i stylizacji potrzebne do wykonania niektórych ćwiczeń. W treści znajdują się informacje kiedy dany plik będzie potrzebny i w jaki sposób należy z niego skorzystać.

Omówienie budowy aplikacji QGIS

Struktura wewnętrzna

QGIS jest napisany w języku programowania C++ i ma budowę modułową. Biblioteki zawierają elementy, z których korzysta aplikacja do wykonywania operacji. Elementy te (tzw. API, *Application Programming Interface*) mogą być zaimportowane z poziomu języka Python i wykorzystane np. do zarządzania danymi przestrzennymi.

Biblioteki QGIS można nazwać platformą programistyczną, na której budowane są właściwe aplikacje takie jak *QGIS Desktop*, *QGIS Server* (publikacja danych w Internecie w formie usług sieciowych WMS/WFS) czy *QField* (mobilna aplikacja na system Android do prac terenowych).

Aplikacja QGIS np. qgis.exe

Biblioteki QGIS:

- qgis.core
- qgis.gui
- qgis.analysis
- natywne sterowniki
- ...

Zewnętrzne biblioteki:

- Qt
- sterowniki GDAL/OGR
- PROJ
- GEOS
- ...

QGIS jako program *Open Source* korzysta z wielu niezależnych projektów, które ułatwiają pisanie i utrzymanie aplikacji. Najważniejsze z nich to:

- **Qt** - pakiet bibliotek i narzędzi, służących głównie tworzeniu wieloplatformowych graficznych interfejsów aplikacji (GUI), wykorzystywany m.in. przy tworzeniu wtyczek,
- **GDAL/OGR** - odczyt i zapis rastrowych (GDAL) i wektorowych (OGR) danych przestrzennych zapisanych w różnych formatach,
- **GEOS** - przetwarzanie danych geometrycznych,
- **PROJ** - transformacja współrzędnych pomiędzy różnymi układami.

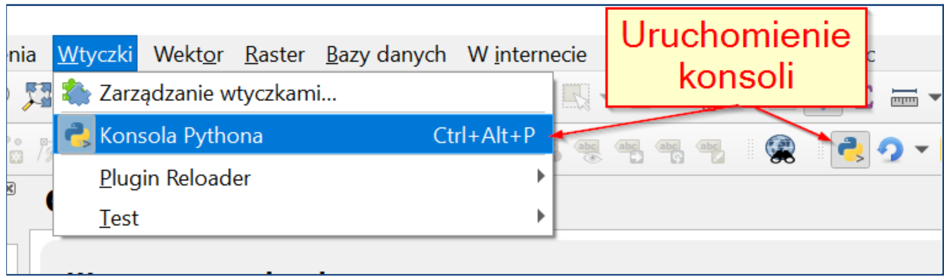
Są one wykorzystywane wewnętrznie w QGIS. Programując w Pythonie nie korzysta się z tych projektów bezpośrednio ale za pomocą bibliotek QGIS API. Szczegółowo zostaną one opisane w rozdziale *Biblioteki QGIS*.

Konsola Python w QGIS

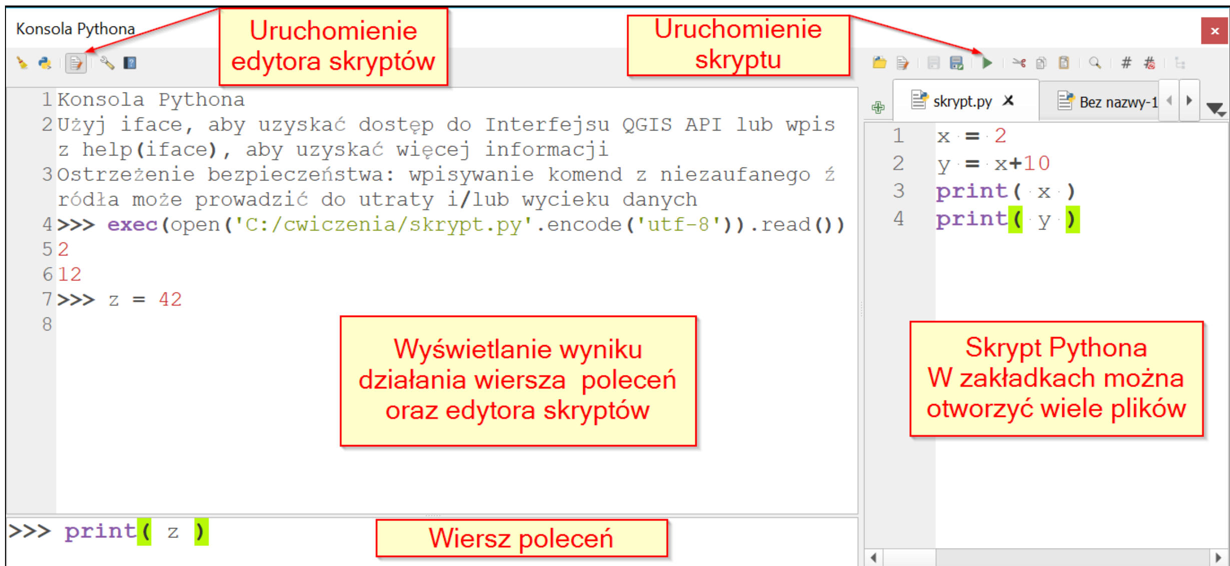
Język programowania Python jest wykorzystywany w środowisku QGIS jako język skryptowy. Dzięki niemu możliwe jest tworzenie nowych lub modyfikowanie działania istniejących narzędzi, zmienianie wyglądu aplikacji czy automatyzacji pracy. Język ten można wykorzystać w kilku miejscach aplikacji. Na szkoleniu skupimy się na dwóch z nich tj. Konsoli Pythona oraz wtyczkach. Poza tym z poziomu Pythona możliwe jest m.in. tworzenie funkcji w kreatorze wyrażeń, akcji warstw czy skryptów dla wtyczki *Processing*.
Wbudowana *Konsola Pythona* to nic innego jak nakładka graficzna na interpreter tego języka. Umożliwia ona pisanie kodu i wywoływanie go w interpreterze w dwóch trybach:

- wiersz poleceń umożliwiający wykonywanie poleceń po wpisaniu ich w konsoli,
- *Edytor skryptów* umożliwiający pisanie i uruchamianie plików z kodem źródłowym.

Oba narzędzia są ze sobą zintegrowane i uruchomione we wspólnym środowisku. Oznacza to, że zmienne, importy itp. zdefiniowane w jednym z tych miejsc są również widoczne w drugim.
W konsoli automatycznie dostępne są klasy QGIS API i nie ma potrzeby ich importowania. Aby uruchomić konsolę należy w QGIS z menu *Wtyczki* wybrać polecenie *Konsola Pythona* lub kliknąć odpowiedni przycisk na pasku narzędzi.
W związku z powyższym konsola Pythona bardzo dobrze nadaje się do nauki operowania danymi w środowisku QGIS oraz testowania różnych rozwiązań, które następnie można wykorzystać w dedykowanych narzędziach jak skrypty czy wtyczki.



Elementy Konsoli



Konsola OSGeo4W Shell

QGIS dostarcza specjalną powłokę *Osgeo4W Shell*, z której możliwy jest dostęp do różnych narzędzi instalowanych razem z tą aplikacją np. interpreter Python, GDAL/OGR, Qt5.

Po uruchomieniu pojawi się konsola systemowa odpowiednio skonfigurowana do wywoływania poleceń dostarczonych razem z instalatorem QGIS. Aby wylistować większość z nich można wywołać poleceniem **`o-help`**.

```
OSGeo4W Shell
run o-help for a list of available commands
C:\OSGeo4W>o-help
                   -={ OSGeo4W Shell Commands }=-
  applygeo                                avcdelete
  avcexport                               avcimport
  bgspawn                                 curl
  dllupdate                               gdaladdo
  gdalbuildvrt                            gdaldem
  gdalenhance                             gdalinfo
  gdallocationinfo                        gdalmanage
  gdalmdiminfo                            gdalmdimtranslate
  gdalsrsinfo                             gdaltindex
  gdaltransform                           gdalwarp
  gdal_contour                            gdal_create
  gdal_grid                               gdal_rasterize
  gdal_translate                          gdal_viewshed
  geotifcp                                getspecialfolder
  gnmanalyse                              gnmmanage
  gpsbabel                                iconv
  las2las                                 las2txt
  lasblock                                lasinfo
  listgeo                                 makegeo
  nearblack                               ogr2ogr
  ogrinfo                                 ogrlineref
  ogrtindex                               osgeo4w-setup-work
  osgeo4w-setup                           psql
  python                                  python3
  pythonw                                 pythonw3
  qgis-ltr-bin                            sqlite3
  testepsg                                textreplace
  ts2las                                  txt2las
  xmlcatalog                              xmllint
  xxmklink                                grass78
  o-help                                  o4w_env
  python-grass78                          python-qgis-ltr
  qgis-ltr-designer                       qgis-ltr
  qgis_process-qgis-ltr                   saga_gui
  setup

GDAL 3.3.1, released 2021/06/28
```

Ilość poleceń może być zmienna w zależności od zainstalowanych dodatkowych aplikacji i bibliotek. Aby uruchomić dane narzędzie należy wpisać jego nazwę. Przykładowo do uruchomienia interpretera języka Python należy wpisać polecenie **`python`**.

```
OSGeo4W Shell - python

C:\OSGeo4W>python
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

W przypadku gdy po wpisaniu polecenia **`python`** pojawi się informacja, że jest ono nieznane lub uruchomi się interpreter w wersji 2, należy dodatkowo wpisać polecenie **`py3_env`**, aby odpowiednio skonfigurować konsolę do korzystania z interpretera w wersji 3. Dotyczy to jednak tylko starszych instalacji QGIS.

QGIS API

API, skrót od *application programming interface* czyli interfejs programowania aplikacji, jest to sposób komunikacji pomiędzy różnymi programami. Dzięki temu możliwe jest wydawanie poleceń z poziomu jednej aplikacji, które są wykonywane przez inny program. Przykładem mogą być zapytania HTTP(S) wysyłane przez przeglądarkę, a przetwarzane przez aplikację serwerową. W naszym przypadku polecenia będą wydawane z poziomu interpretera Pythona, a wykonywane przez QGIS.

QGIS posiada własny zbiór bibliotek zawierających definicje różnych obiektów wykorzystywanych w trakcie działania aplikacji. Składają się one na tzw. QGIS API, gdzie są zdefiniowane zarówno elementy dotyczące danych przestrzennych (m.in. warstwy, obiekty, geometrie) jak i graficznego interfejsu użytkownika (m.in. przyciski, panele, okno mapy). QGIS API składa się w całości z klas reprezentujących konkretne obiekty np. warstwę, przycisk, geometrię, układ współrzędnych. Każdy z tych elementów ma własną klasę opisaną w dokumentacji. Dostępne są dwie wersje dokumentacji:

- **http://www.qgis.org/api** - opis klas i funkcji QGIS dla C++
- **http://www.qgis.org/pyqgis** - dokumentacja dla Pythona

Na obu stronach możliwe jest wybranie konkretnej wersji QGIS: C++ od 1.6, Python od 3.0. Wersja *master* odnosi się do aktualnie rozwijanej wersji QGIS, może więc zawierać elementy, które się zmienią w przyszłości, dlatego nie jest zalecane korzystanie z tej wersji dokumentacji. Najlepiej wybrać dokumentację dla wersji QGIS, z której korzysta się na co dzień w pracy albo ostatnią wersję LTR (o wydłużonym wsparciu), ale nie starszą niż 3.10 ponieważ od tej wersji dokumentacja została poprawiona i jest łatwiejsza w użytkowaniu.

Na szkoleniu korzystać będziemy z dokumentacji dla języka Python: http://www.qgis.org/pyqgis**.** Strona główna zawiera wyszukiwarkę oraz spis wszystkich dostępnych klas z podziałem na moduły, w których się znajdują. Po wejściu w dokumentację danej klasy znajdziemy w niej krótki jej opis oraz spis metod (funkcji) i atrybutów (zmiennych) dostępnych z jej poziomu. Po kliknięciu w nazwę można przejść do szczegółowszych informacji dotyczących danego elementu klasy np. jakie argumenty przyjmuje metoda i co zwraca jako wynik działania.

Przykład opisu klasy w dokumentacji QGIS API dla języka Python. **`QgsPointXY`** - klasa reprezentująca punkt w przestrzeni 2D:

Class: QgsPointXY — Nazwa klasy

class qgis.core.QgsPointXY — Informacje o module

Bases: sip.wrapper

QgsPointXY(p: QgsPointXY) Create a point from another point

QgsPointXY(x: float, y: float) Create a point from x,y coordinates — Sposoby tworzenia instancji klasy

Parameters:
- x – x coordinate
- y – y coordinate

Parametry potrzebne do stworzenia instancji

Zgodnie z dokumentacją klasa **QgsPointXY** znajduje się w module **qgis.core**. Instancję tej klasy można stworzyć na kilka sposobów m.in. podając dwie liczby reprezentujące współrzędne XY lub inną instancję tej klasy (nastąpi klonowanie, czyli wszystkie ustawienia (atrybuty) zostaną skopiowane, ale powstanie nowy, niezależny obiekt). Ten drugi sposób jest często spotykany w QGIS API i służy kopiowaniu obiektów.

```
# Import klasy
from qgis.core import QgsPointXY

# Stworzenie instancji na podstawie współrzędnych
punkt1 = QgsPointXY( 10, 15 )

# Stworzenie instancji na podstawie istniejącej instancji tej samej klasy (klonowanie), punkt2 będzie miał te same współrzędne co punkt1
punkt2 = QgsPointXY( punkt1 )
```

Opis metod w dokumentacji:

Nazwa — Argumenty

azimuth(*self*, *other*: *QgsPointXY*) → float — Typ danych wyjściowych

Calculates azimuth between this point and other one (clockwise in degree, starting from north) — Opis metody

Parameters: other (*QgsPointXY*) – — Lista argumentów

Return type: float — Zwracana wartość

Większość metod jako pierwszy argument ma podany obiekt **self**. Jest to odniesienie do instancji danej klasy, które jest automatycznie podawane przez Python, więc należy ją pominąć przy wywoływaniu metody.

```
# Ustawienie współrzędnej X
punkt1.setX( 20.5 )
print( punkt1.x() )
# 20.5
print( punkt2.x() )
# 10
```

Biblioteki QGIS

QGIS API zorganizowane jest w kilku bibliotekach. Z poziomu Python najczęściej wykorzystywane są następujące moduły:

- **`qgis.core`** - biblioteka core zawiera wszystkie podstawowe funkcje GIS (m.in. obsługa warstw, projektu, akcji),
- **`qgis.gui`** - zawiera graficzne widżety, pozwala na interakcję z oknem QGIS,
- **`qgis.analysis`** - biblioteka ułatwiająca operacje geometryczne na warstwach i obiektach, tworzenie grafów, operacje sieciowe (wykorzystuje narzędzia z modułu **`qgis.core`**).

Import obiektów z powyższych klas można wykonać w następujący sposób:

```
from qgis.core import <nazwa_klasy>
```

Główne klasy QGIS API

Klasy QGIS API (z nielicznymi wyjątkami) rozpoczynają się od przedrostka **`Qgs`**, po którym następuje właściwa nazwa klasy.

qgis.core

- **`QgsMapLayer`** – klasa bazowa dla wszystkich rodzajów warstw
- **`QgsRasterLayer`**, **`QgsVectorLayer`** – klasy reprezentujące odpowiednio warstwę rastrową i wektorową, niezależnie od formatu danych źródłowych
- **`QgsRasterDataProvider`**, **`QgsVectorDataProvider`** – klasy bazowe dla sterowników warstw rastrowych i wektorowych, każdy sterownik (*ogr*, *postgres*, *spatialite* itd.) posiada własną implementację tych klas, służą do komunikacji ze źródłem danych (odczyt, zapis)
- **`QgsFeature`** – klasa reprezentująca obiekt warstwy
- **`QgsFields`** – zawiera wszystkie pola (lista obiektów **`QgsField`**) danej warstwy wektorowej
- **`QgsField`** – klasa przechowująca informacje o polu (kolumnie) tabeli atrybutów m.in. typ danych, długość, nazwa
- **`QgsGeometry`** – geometria obiektu
- **`QgsWkbTypes`** - przechowuje informacje o typach geometrii
- **`QgsPointXY`** – klasa reprezentująca punkt 2D

- **`QgsRectangle`** - klasa reprezentująca prostokąt określony współrzędnymi min_x, min_y, max_x, max_y

qgis.gui

- **`QgisInterface`** – specjalna klasa umożliwiająca komunikację pomiędzy Pythonem, a uruchomionym środowiskiem QGIS,
- **`QgsMapCanvas`** – okno mapy,
- **`QgsMessageBar`** - pozwala wyświetlać informacje użytkownikowi nad oknem mapy
- **`QgsMapLayerComboBox`** - pole wyboru warstwy z listy wczytanych do QGIS, automatycznie aktualizowane przy dodawaniu/usuwaniu warstw.
- **`QgsFieldComboBox`** - pole wyboru pola tabeli atrybutów danej warstwy wektorowej, automatycznie aktualizowane przy zmianie warstwy.
- **`QgsMapTool`** - klasa bazowa dla narzędzi mapy QGIS,
- **`QgsMapToolEmitPoint`** - klasa reagująca na klikanie po mapie,
- **`QgsRubberBand`** - umożliwia rysowanie po mapie.

Ćwiczenie

Treść zadania

Korzystając z dokumentacji QGIS API zaimportuj klasę **`QgsProject`** w *Konsoli Pythona*.

Opis

Po uruchomieniu *Konsoli Pythona* należy wpisać polecenie:

```
from qgis.core import QgsProject
```

i wcisnąć przycisk *Enter*.

```
Konsola Pythona
1 Konsola Pythona
2 Użyj iface, aby uzyskać dostęp do Interfejsu QGIS API lub wpisz help(iface), aby uzyskać więcej informacji
3 Ostrzeżenie bezpieczeństwa: wpisywanie komend z niezaufanego źródła może prowadzić do utraty i/lub wycieku danych
4 >>> from qgis.core import QgsProject
5
>>>
```

Obiekt *iface*

`iface` jest instancją klasy **`QgisInterface`**. Jest to specjalny obiekt, który umożliwia z poziomu języka Python na sterowanie oknem uruchomionej aplikacji QGIS. Dzięki niej możliwe jest m.in. wczytywanie warstw, pobranie warstwy aktywnej, dodanie paneli dokowanych, rejestracja nowych pasków narzędzi wraz z przyciskami czy dodawanie menu.

Istnieje tylko jedna instancja klasy **QgisInterface**, nie jest możliwe jej utworzenie, a jedynie pozyskanie z QGIS. Jest on domyślnie dostępny w *Konsoli Pythona* oraz we wtyczkach, ale możliwe jest jego zaimportowanie w dowolnym miejscu:

```
from qgis.utils import iface
```

Wybrane metody obiektu **iface**:

```
# Zwraca aktywną warstwę w QGIS
iface.activeLayer()
# <QgsMapLayer: 'nazwa' (ogr)>

# zwraca okno mapy QGIS (QgsMapCanvas)
iface.mapCanvas()
# <qgis._gui.QgsMapCanvas object at 0x...>
```

Ćwiczenie

Treść zadania

Wykorzystując dokumentację QGIS API znajdź metodę klasy **QgsMapCanvas**, która zwraca zasięg widocznej mapy i wydrukuj wynik działania tej metody.

Opis

Główne okno mapy QGIS to instancja klasy **QgsMapCanvas**. Aby ją zwrócić należy skorzystać z obiektu **iface** i jego metody **mapCanvas()**. W dokumentacji należy zlokalizować metodę, która zwraca zasięg widoku mapy. Jest to metoda **extent**, która nie przyjmuje żadnych argumentów. Wynikiem jest klasa **QgsRectangle** reprezentująca prostokątny obszar mapy ze współrzędnymi minimalnymi (dolny lewy wierzchołek) i maksymalnymi (górny prawy wierzchołek). Zwrócone wartości mogą być różne od podanych poniżej i są uzależnione od wielu czynników takich jak przybliżenie i przesunięcie mapy, kształt okna mapy, widoczność i wielkość paneli, pasków narzędzi itd.

Kod źródłowy

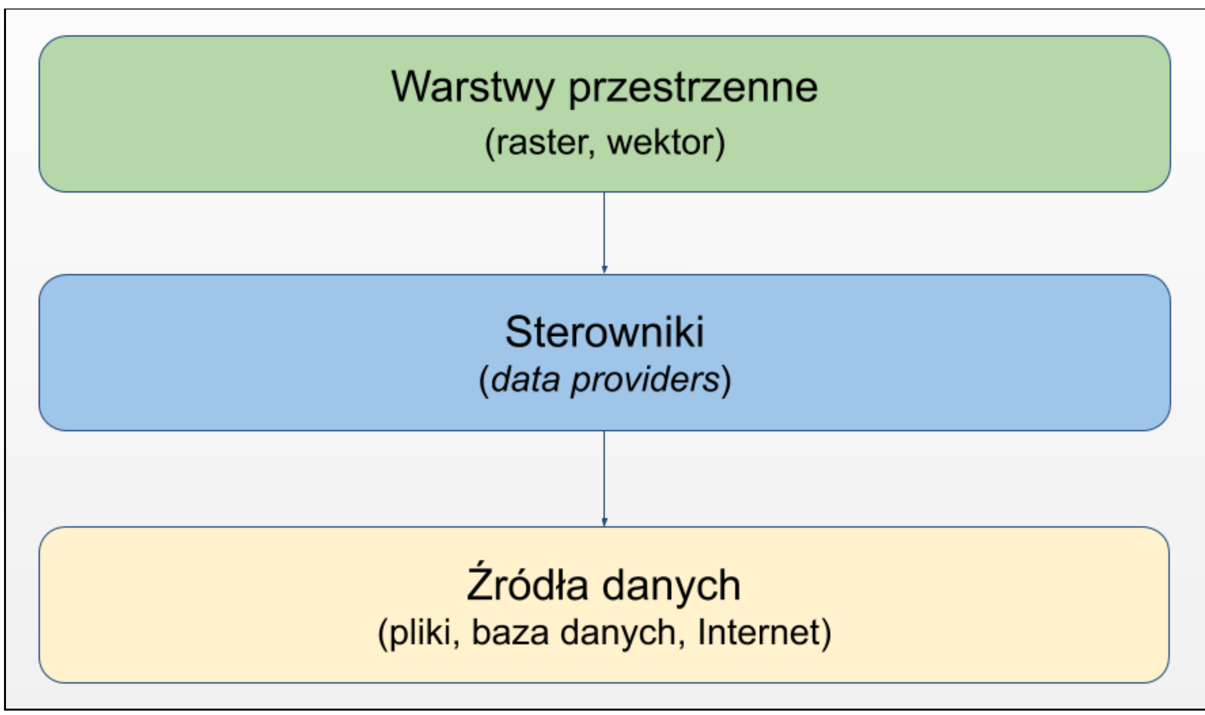
```
iface.mapCanvas().extent()
# <QgsRectangle: 598233.1568823215784505 171788.75010274662054144,
606720.57867100543808192 174552.93157954452908598>
```

Obsługa warstw przestrzennych

QGIS wspiera wiele formatów danych przestrzennych. Do ich obsługi wykorzystywane są tzw. sterowniki (*data providers*), które odpowiadają za komunikację ze źródłem danych (plikiem, bazą danych). Sterowniki odczytując informacje ze źródła konwertują je do ujednoliconej formy warstw przestrzennych QGIS, dzięki temu niezależnie od formatu każda

warstwa jest obsługiwana przez QGIS w ten sam sposób. Odpowiadają one również za zapisywanie zmian.

Każdy typ warstwy przestrzennej ma własną klasę, która go reprezentuje. W przypadku warstw rastrowych jest to **`QgsRasterLayer`**, a wektorowych **`QgsVectorLayer`**. Wszystkie rodzaje warstw dziedziczą z klasy bazowej **`QgsMapLayer`**, w której zebrane są wspólne cechy m.in. układ współrzędnych, zasięg, nazwa wyświetlana czy unikalny identyfikator. Informacje specyficzne dla danego typu warstw są natomiast opisane w dedykowanych klasach np. **`QgsVectorLayer`** przechowuje informacje o obiektach przestrzennych, schemacie tabeli atrybutów i typie geometrii, a **`QgsRasterLayer`** o liczbie kanałów czy rozdzielczości komórek rastra.



Do pobrania aktywnej warstwy z poziomu Pythona należy wykorzystać obiekt iface:

```
warstwa = iface.activeLayer()
print( type(warstwa) )
# <class 'qgis._core.QgsVectorLayer'>
```

Z każdą warstwą przestrzenną powiązany jest jeden sterownik, uzależniony od formatu źródła danych. Dostęp do sterownika z poziomu warstwy można uzyskać za pomocą metody **`dataProvider()`**.

```
sterownik = warstwa.dataProvider()
print( type(sterownik) )
# <class 'qgis._core.QgsVectorDataProvider'>
```

Dane rastrowe

Warstwy rastrowe składają się z komórek (pikseli) przyjmujących wartości numeryczne. Każdy raster składa się z jednego lub więcej kanałów, które numerowane są kolejnymi liczbami naturalnymi, gdzie indeks 1 ma kanał pierwszy, 2 kanał drugi itd.
Klasą reprezentującą dane rastrowe jest **`QgsRasterLayer`**. Główne metody tej klasy:

- **`width()`** - szerokość rastra w pikselach,
- **`height()`** - wysokość rastra w pikselach,
- **`rasterUnitsPerPixelX()`** - szerokość piksela w jednostkach układu współrzędnych warstwy,
- **`rasterUnitsPerPixelY()`** - wysokość piksela w jednostkach układu współrzędnych warstwy,
- **`bandCount()`** - liczba kanałów,
- **`dataProvider()`** - sterownik danych rastrowych, zwraca instancję klasy **`QgsRasterDataProvider`.**

Przykłady użycia, *raster* - instancja klasy **`QgsRasterLayer`**:

```
raster.width()
# 44
raster.unitsPerPixelX()
# 0.25
raster.bandCount()
# 36
raster.dataProvider()
# <qgis._core.QgsRasterDataProvider object at 0x... >
```

Jeśli wymagane jest pobranie wartości komórek rastra należy skorzystać z klasy **`QgsRasterDataProvider`** i jej metod:

- **`sourceNoDataValue( numer_kanalu )`** - liczba oznaczająca brak danych rastra w danym kanale

```
raster.dataProvider().sourceNoDataValue(1)
# -9999.0
```

- **`bandStatistics( numer_kanalu )`** - obliczenie różnych statystyk dla danego kanału np. średnia wartość komórek, najmniejsza/największa wartość, odchylenie standardowe itd. Komórki oznaczone jako brak danych nie są brane pod uwagę przy obliczeniach. Metoda zwraca instancję klasy **`QgsRasterBandStats`**, której atrybuty przechowują obliczone wartości.

```
# Sterownik warstwy rastrowej (QgsRasterDataProvider)
sterownik = raster.dataProvider()
```

```
# Obliczenie statystyk kanału pierwszego
statystyki = sterownik.bandStatistics( 1 )
# mean - wartość średnia wszystkich komórek rastra
print( statystyki.mean )
# 173
```

- **identify(punkt, format)** - odczyt wartości komórki rastra w danym punkcie (**QgsPointXY**) i formacie. Jako format należy zawsze podawać **QgsRaster.IdentifyFormatValue** aby uzyskać informacje jako słownik Pythona. Pozostałe formaty są zarezerwowane dla QGIS. Metoda zwraca instancję klasy **QgsRasterIdentifyResult**, której metoda **results()** pozwala na dostęp do danych poprzez słownik którego kluczem jest numer kanału, a wartością odczytana wartość komórki w tym kanale. Jeśli w danym punkcie jest brak wartości lub jest on poza zasięgiem rastra to zwracany jest dla danego kanału obiekt **None**.

```
# Punkt analizy
punkt = QgsPointXY(20, 50)
# Sterownik warstwy rastrowej (QgsRasterDataProvider)
sterownik = raster.dataProvider()
# Odczyt wartości komórek w podanym punkcie
wartosc = sterownik.identify( punkt, QgsRaster.IdentifyFormatValue )
print( wartosc.results() )
# { 1 : 872.0, 2 : 304, 2 : None... }
```

Dane wektorowe

Warstwę wektorową tworzą obiekty przestrzenne składające się z geometrii i atrybutów. Każda warstwa ma zdefiniowany typ geometrii (punkty, poligony wieloczęściowe, brak geometrii itp.) oraz schemat tabeli atrybutów.
Klasą reprezentującą dane wektorowe jest **QgsVectorLayer**. Główne metody tej klasy:

- **geometryType()** - **QgsWkbTypes.GeometryType**, uproszczony typ geometrii
- **wkbType()** - **QgsWkbTypes.Type**, szczegółowy typ geometrii
- **fields()** - schemat tabeli atrybutów (**QgsFields**)
- **getFeatures()** - iteracja po obiektach warstwy

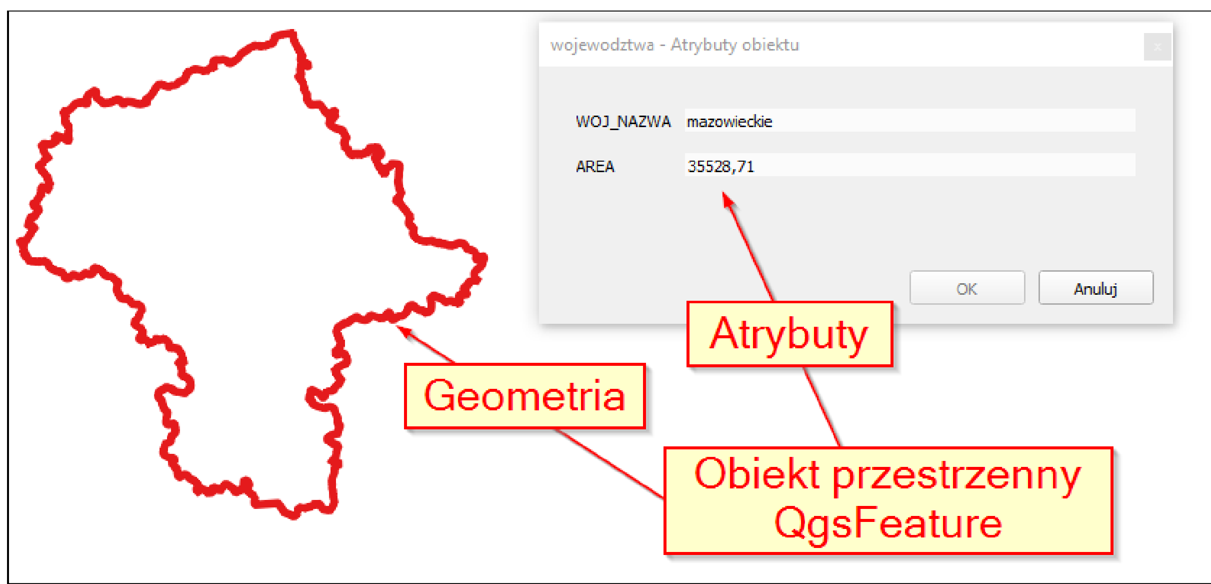
Przykłady użycia, *wektor* - instancja klasy **QgsVectorLayer**:

```
wektor.featuresCount()
# 16
wektor.crs()
# <qgis._core.QgsCoordinateReferenceSystem object at 0x... >
wektor.fields()
# <qgis._core.QgsFields object at 0x... >
```

```
# Iteracja po wszystkich obiektach
for obiekt in wektor.getFeatures():
    print( obiekt )

# Iteracja po zaznaczonych obiektach
for obiekt in wektor.getSelectedFeatures():
    print( obiekt )

# Pobranie pojedynczego obiektu o ID 0:
obiekt = wektor.getFeature( 0 )
```

Obiekty przestrzenne

Pojedynczy obiekt przestrzenny warstwy wektorowej reprezentowany jest klasą **QgsFeature**. Przechowuje on informacje o geometrii i wartościach atrybutów danego rekordu oraz jego unikalny identyfikator w obrębie warstwy.



Główne metody:

- **id()** - unikalny identyfikator obiektu (liczba całkowita),
- **fields()** - schemat tabeli atrybutów (**QgsFields**),
- **geometry()** - zwraca geometrię obiektu (**QgsGeometry**),
- **setGeometry**(**geometria**) - ustawia geometrię obiektu, geometria → **QgsGeometry**,
- **attributes()** - lista wartości atrybutów,
- **setAttributes**(**atrybuty**) - lista wartości atrybutów, atrybuty - lista wartości do wstawienia.

```
for obiekt in warstwa.getFeatures():
    print( obiekt.id() )
```

```
    # 0
    print( obiekt.attributes() )
    # [ 'tekst1', 19935.94 ]
```

Aby odczytać wartość pojedynczego atrybutu obiektu przestrzennego można skorzystać z metody attribute lub wykorzystując indeksację. W obu przypadkach możliwe jest podanie nazwy lub indeksu atrybutu:

```
# Wartość atrybutu po jego nazwie
obiekt.attribute( 'nazwa' )
obiekt['nazwa']

# Wartość atrybutu po indeksie pola
obiekt.attribute( 2 )
obiekt[2]
```

Modyfikacja istniejącej warstwy

Modyfikacja danych na warstwie przestrzennej odbywa się za pomocą sterownika warstwy (**`QgsVectroDataProvider`**), który można zwrócić dla danej warstwy wektorowej za pomocą metody **`dataProvider()`** - analogicznie jak dla warstwy rastrowej. Zmiany są zapisywane bezpośrednio w źródle, aby je zobaczyć na wczytanej warstwie przestrzennej w QGIS należy ją odświeżyć za pomocą metody **`reload()`**:

```
warstwa.reload()
```

Do zmian należy wykorzystać odpowiednie metody tej klasy:

- **`addFeatures`** - jako argument należy podać listę instancji klas **`QgsFeature`**. Ważne jest, aby miały one atrybuty oraz typ geometrii zgodne z warstwą, do której są dodawane.

```
# Dodanie dwóch obiektów
warstwa.dataProvider().addFeatures( [obiekt1, obiekt2] )
```

- **`deleteFeatures`** - należy podać listę identyfikatorów obiektów do usunięcia:

```
# Usunięcie dwóch obiektów
warstwa.dataProvider().deleteFeatures( [ 1, 2 ] )
```

- **`changeGeometryValues`** - zmiana geometrii obiektów, należy znać ich identyfikatory - są one kluczem słownika przechowującego nowe dane:

```
# Zmiana geometrii dla obiektu o ID 1
warstwa.dataProvider().changeGeometryValues( { 1 : geometria } )
```

- **changeAttributeValues** - zmiana wartości atrybutów, należy podać indeksy pól, których wartości mają być modyfikowane:

```
atrybuty = {
    0 : 20,     # Pierwsze pole
    1 : 'opis'  # Drugie pole
}
# Zmiana wartości dwóch pól w obiekcie o ID 1
warstwa.dataProvider().changeAttributeValues( { 1 : atrybuty } )
```

- **changeFeatures** - edycja obiektów (geometrii i atrybutów) - łączy funkcjonalność metod **changeGeometryValues** i **changeAttributeValues**:

```
# Zmiana atrybutów jednego obiektu i geometrii dwóch obiektów
warstwa.dataProvider().changeFeatures(
    { 1 : atrybuty }, # słownik z atrybutami
    { 1 : geometria1, 2 : geometria2 } )    # słownik z geometriami
```

- **addAttributes** - dodanie nowych kolumn do warstwy, należy podać listę instancji klasy **QgsField**:

```
# Dodanie dwóch kolumn
warstwa.dataProvider().addAttributes( [
    QgsField( "pole_tekstowe", QVariant.String ),
    QgsField( "pole_liczbowe", QVariant.Int )
] )
```

- **deleteAttributes** - usunięcie kolumn, należy podać listę indeksów kolumn do usunięcia:

```
#Usunięcie dwóch pierwszych kolumn
warstwa.dataProvider().deleteAttributes( [ 0, 1 ] )
```

W powyższych metodach aby określić atrybut do zmiany należy podać jego indeks. Często jednak zdarza się, że dysponujemy jedynie jego nazwą. W takim wypadku należy sprawdzić jaki indeks ma pole o danej nazwie. Pomocna tu jest klasa **QgsFields**, która przechowuje informacje o schemacie tabeli atrybutów danej warstwy, w tym o ich kolejności. Jedną z metod tej klasy jest **indexFromName**, gdzie jako argument podajemy nazwę pola, jeśli jest ono w tabeli to zostanie zwrócony jego indeks, jeśli nie otrzymamy *-1*.

```
# Pobranie schematu tabeli warstwy wektorowej
tabela = warstwa.fields()
# Indeks pola o podanej nazwie
indeks_pola = tabel.indexFromName( 'nazwa' )
```

Ćwiczenie

Treść zadania

Edytuj warstwę *punkty* wstawiając w pole *Wysokosc* losową liczbę z zakresu 1-2 w dokładnością do jednego miejsca po przecinku.

Opis

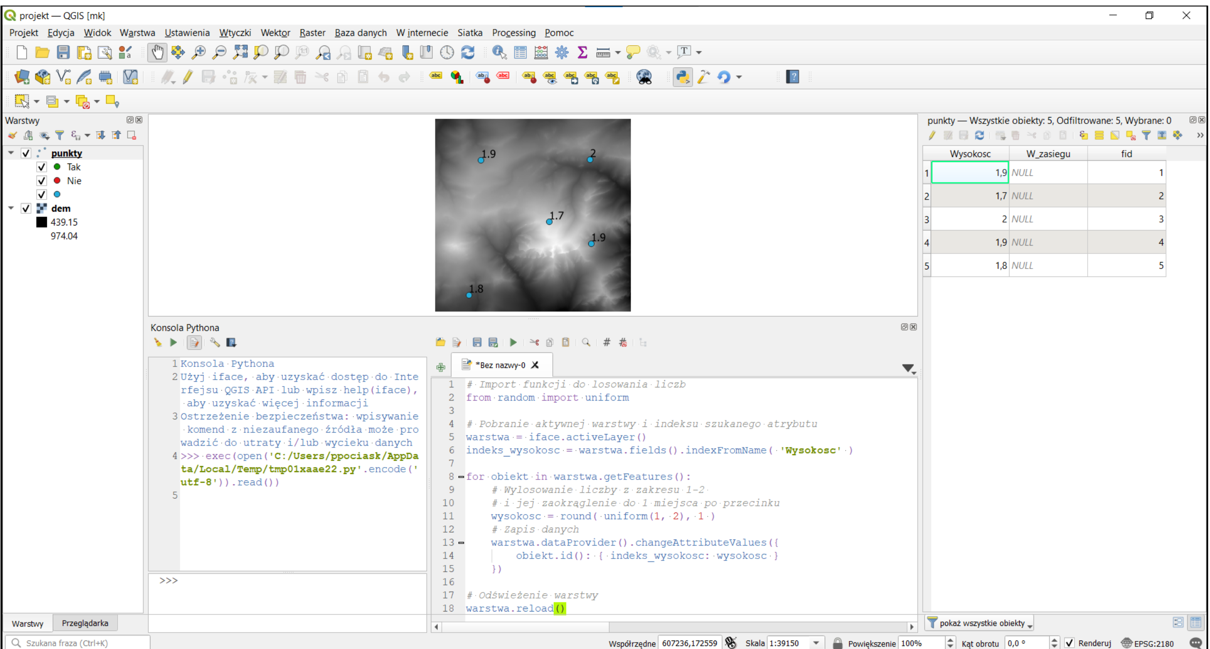
Aby uzyskać dostęp do warstwy *punkty* należy ją zaznaczyć w panelu legendy i wywołać w konsoli polecenie **`warstwa=iface.activeLayer()`**. W ten sposób pod zmienną warstwa mamy instancję klasy **`QgsVectorLayer`**.

Do modyfikacji wartości konkretnego atrybutu z poziomu sterownika potrzebny jest indeks pola w schemacie. W tej chwili dysponujemy tylko nazwą tej kolumny. Aby uzyskać jej indeks należy wywołać metodę **`indexFromName()`** klasy **`QgsFields`** reprezentującej schemat warstwy.

W kolejnym etapie należy wykonać iterację po obiektach przestrzennych warstwy aby uzyskać informacje o ich identyfikatorze i powierzchni. Moduł **`random`** jest biblioteką służącą do zwracania liczb pseudolosowych. Dostępnych jest szereg metod, z których **`uniform`** pozwala określić zasięg i zwracana jest liczba zmiennoprzecinkowa typu **`float`**. Po wylosowaniu liczbę trzeba zaokrąglić do 1 miejsca po przecinku za pomocą funkcji **`round`**.

Aby zaktualizować wartość atrybutu danego obiektu należy skorzystać ze sterownika warstwy i jego metody **`changeAttributeValues`**. Argumentem jest słownik, którego kluczami są identyfikatory edytowanych obiektów, a wartością kolejny słownik. W tym słowniku podajemy jako klucz indeks pola, które chcemy zmodyfikować w podanym obiekcie a wartością jest wylosowana wcześniej liczba.

Na końcu możemy odświeżyć informacje o zmianach w źródle danych za pomocą metody **`reload()`**.



Kod źródłowy

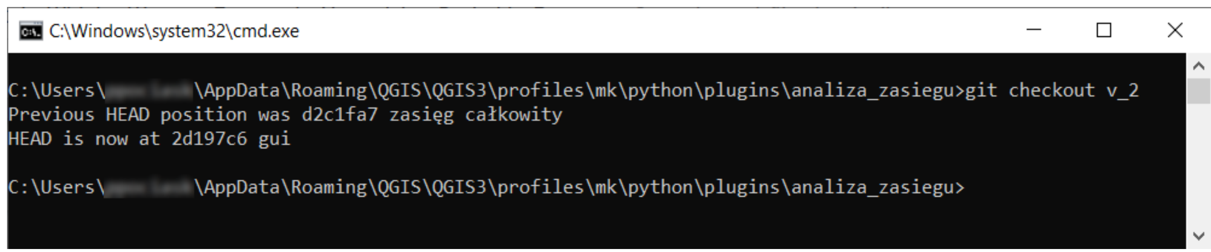
```python
# Import funkcji do losowania liczb
from random import uniform

# Pobranie aktywnej warstwy i indeksu szukanego atrybutu
warstwa = iface.activeLayer()
indeks_wysokosc = warstwa.fields().indexFromName( 'Wysokosc' )

for obiekt in warstwa.getFeatures():
    # Wylosowanie liczby z zakresu 1-2
    # i jej zaokrąglenie do 1 miejsca po przecinku
    wysokosc = round( uniform(1, 2), 1 )
    # Zapis danych
    warstwa.dataProvider().changeAttributeValues({
        obiekt.id(): { indeks_wysokosc: wysokosc }
    })

# Odświeżenie warstwy
warstwa.reload()
```

GIT i systemy kontroli wersji

GIT to tzw. system kontroli wersji. Oprogramowanie tego typu służy programistom do zarządzania zmianami w kodzie źródłowym. Umożliwia śledzenie prac, przeglądanie historii, tworzenie niezależnych wersji kodu i późniejsze ich scalanie itp.
Na potrzeby szkolenia możliwe jest skorzystanie z systemu GIT w celu przełączania kodu źródłowego do wersji, która jest aktualnie omawiana na zajęciach. Dla ułatwienia aplikacja ta jest dołączona do kodu źródłowego z katalogu ćwiczeniowego i w każdej chwili uczestnik może włączyć odpowiednią wersję. Nie ma potrzeby instalacji dodatkowego oprogramowania.
Aby uruchomić GIT należy wejść w katalog wtyczki i podkatalog *git* i uruchomić plik *start.bat*. Pojawi się konsola systemowa, w której można wpisywać polecenia GIT. Katalog roboczy jest automatycznie ustawiony na katalog wtyczki.
Aby przełączyć się pomiędzy wersjami wtyczki należy wywołać polecenie **`git checkout <wersja>`**, gdzie jako wersja należy podać odpowiednią nazwę. Przy każdym ćwiczeniu, w którym modyfikowany jest kod wtyczki, podane są polecenia, które należy wpisać w konsoli aby włączyć odpowiednią wersję kodu źródłowego. Przykładowo, jeśli w tekście jest polecenie:

```
git checkout v_3
```

użytkownik może skopiować to polecenie bezpośrednio do konsoli i po wciśnięciu *Enter* kod zostanie zaktualizowany do wersji **`v_3`**.



Wtyczki QGIS

Wykorzystując Python możliwe jest tworzenie wtyczek do QGIS, które dodają nowe funkcje. Dają one niemal pełną kontrolę nad aplikacją, dzięki czemu można stworzyć niemal dowolne narzędzia ułatwiające codzienną pracę. Wbudowany system repozytoriów pozwala udostępniać wtyczki pomiędzy użytkownikami. Domyślnie dostępne jest oficjalne repozytorium, do którego użytkownicy mogą dodawać własne rozszerzenia. Można również tworzyć własne repozytoria. Do zarządzania rozszerzeniami służy *Menedżer wtyczek*.

Wtyczki QGIS objęte są licencją GNU GPL w wersji 2.x lub nowszej. Rozpowszechniając wtyczkę autor zobowiązany jest dostarczyć użytkownikowi również jej kod źródłowy.

Katalog wtyczek można znaleźć wybierając w menu QGIS *Ustawienia -> Profile użytkownika -> Otwórz katalog aktywnego profilu* i przechodząc do katalogu *python/plugins*. Znajdują się tu wszystkie pobrane przez użytkownika rozszerzenia. Tworząc własną wtyczkę, należy skopiować ją do tego katalogu. Aby była ona widoczna w *Menadżerze wtyczek* należy zrestartować aplikację QGIS.
Wtyczka jest widoczna tylko w profilu, do którego została dodana. W pozostałych profilach należy ją zainstalować/skopiować indywidualnie.

W trakcie kursu zostanie przygotowana wtyczka sprawdzająca czy wskazany przez użytkownika punkt znajduje się w zasięgu nadajników zdefiniowanych jako warstwa punktowa.

Plugin Builder

Plugin Builder to narzędzie upraszczające tworzenie wtyczek poprzez wygenerowanie podstawowych plików i zasobów wymaganych przez QGIS. Ułatwia to rozpoczęcie pracy nad nowym rozszerzeniem. Wtyczka ta działa w formie kreatora, gdzie przechodząc przez kolejne okna ustawiane są podstawowe informacje dotyczące metadanych i wyglądu wtyczki.
Plugin Builder dostępny jest w oficjalnym repozytorium QGIS. Przed przystąpieniem do dalszych ćwiczeń należy ją zainstalować za pomocą Menedżera wtyczek QGIS (menu *Wtyczki -> Zarządzanie wtyczkami*).

Ćwiczenie

Wykorzystując wtyczkę *Plugin Builder* stwórz szablon wtyczki, która będzie posiadała panel dokowany.

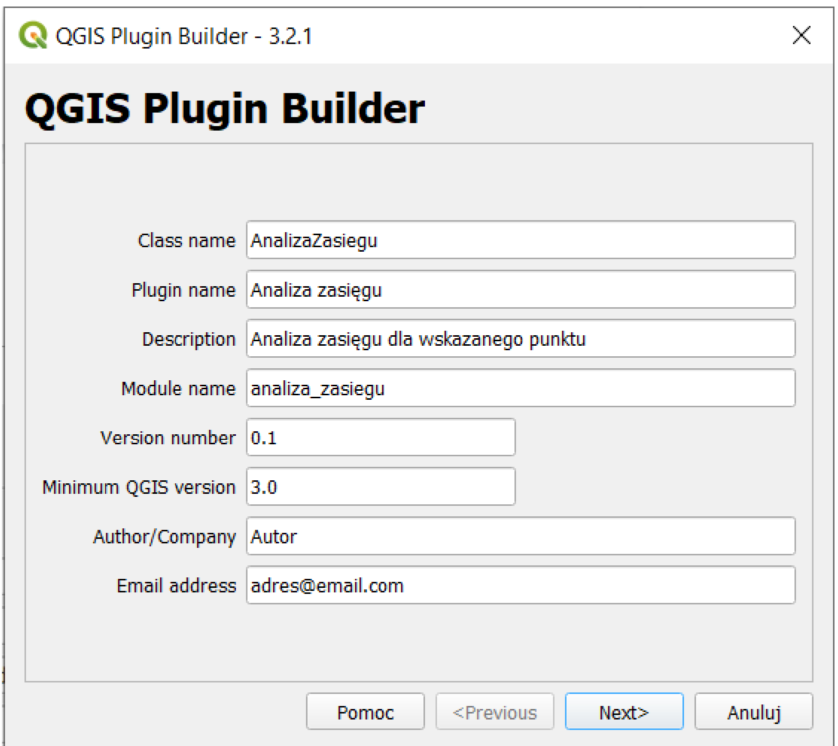
```
git checkout v_1
```

Opis

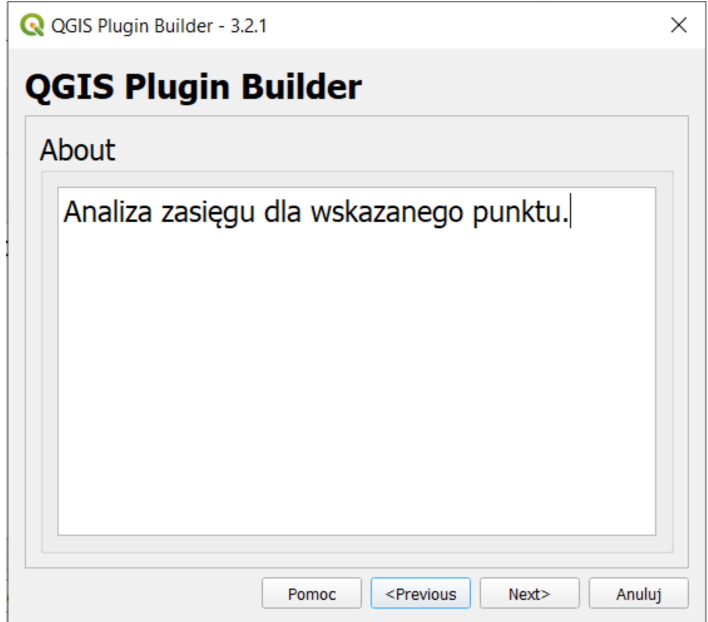
Należy uruchomić kreator wtyczek za pomocą ikony i wypełnić formularze dostępne w kolejnych oknach dialogowych.

Metadane wtyczki

W dwóch pierwszych oknach kreatora ustawiamy metadane wtyczki, które będą wykorzystane m.in. do nazywania utworzonych plików, obiektów programistycznych oraz wyświetlą się w oknie Menedżera wtyczek.



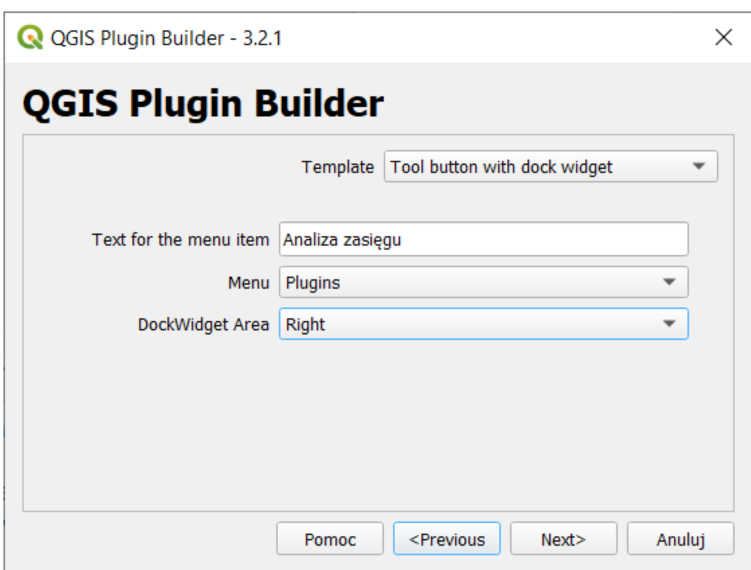
- **Class name** - nazwa klasy reprezentującej w QGIS daną wtyczkę. Nazwa nie może zawierać spacji oraz znaków specjalnych.
- **Plugin name** - Nazwa wtyczki, wyświetlana m.in. w Menedżerze wtyczek i menu.
- **Description** - krótki, jednolinijkowy opis wtyczki wyświetlany w Menedżerze wtyczek.
- **Module name** - nazwa modułu zawierającego klasę wtyczki. Nie powinna ona zawierać spacji oraz znaków specjalnych.
- **Version number** - wersja wtyczki.
- **Minimum QGIS version** - minimalna wersja QGIS wymagana do użytkownika wtyczki, głównie ze względu na używane API.
- **Author/Company** - autor lub nazwa firmy, która stworzyła wtyczkę.
- **Email address** - adres poczty elektronicznej autora.



- **About** - dłuższy opis funkcjonalności wtyczki.

Rodzaj okna dialogowego

W tym kroku określamy rodzaj szablonu wtyczki (lista *Template*). Ma on wpływ na to w jaki sposób zostanie utworzone główny formularz wtyczki (okno dialogowe, panel dokowany lub jego brak). Ma to duży wpływ na sposób w jaki wtyczka będzie działać więc warto dobrze przemyśleć sposób jej funkcjonowania, ponieważ późniejsza zmiana może wiązać się z koniecznością wprowadzenia dużych zmian w kodzie.

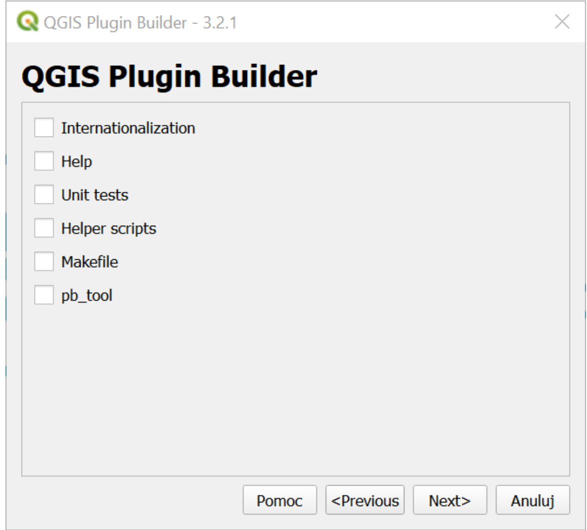


- **Tool button with dialog** - przycisk na pasku narzędzi, który wyświetla osobne okno dialogowe wtyczki.
    - **Text for the menu item** - tekst wyświetlany w menu wtyczki przy przycisku uruchamiającym okno dialogowe.
    - **Menu** - nazwa menu, w którym pojawi się menu wtyczki.

- **Tool button with widget** - przycisk na pasku narzędzi, który wyświetla dokowalne okno dialogowe. Zawiera te same pozycje co poprzedni szablon oraz dodatkowo:
  - **DockWidget Area** - domyślna pozycja okna w stosunku do okna mapy.
- **Processing Provider** - dodanie pozycji w Narzędziach geoprocessingu.
  - **Algorithm name** - nazwa algorytmu.
  - **Algorithm group** - nazwa grupy, w której znajdować się będzie algorytm.
  - **Provider name** - nazwa źródła danych.
  - **Provider description** - opis źródła danych.

Dodatkowe elementy

W tym miejscu możliwe jest określenie czy do wtyczki zostaną dodane dodatkowe pliki. Ma to sens jedynie w przypadku bardziej zaawansowanych narzędzi (m.in. wielojęzycznego interfejsu, dokumentacji wbudowanej we wtyczkę, tworzenia testów jednostkowych). W naszych ćwiczeniach nie będziemy korzystać z tych elementów więc należy odznaczyć wszystkie opcje. Ułatwi to nam również późniejsze zapoznanie się z kluczowymi elementami wtyczki.

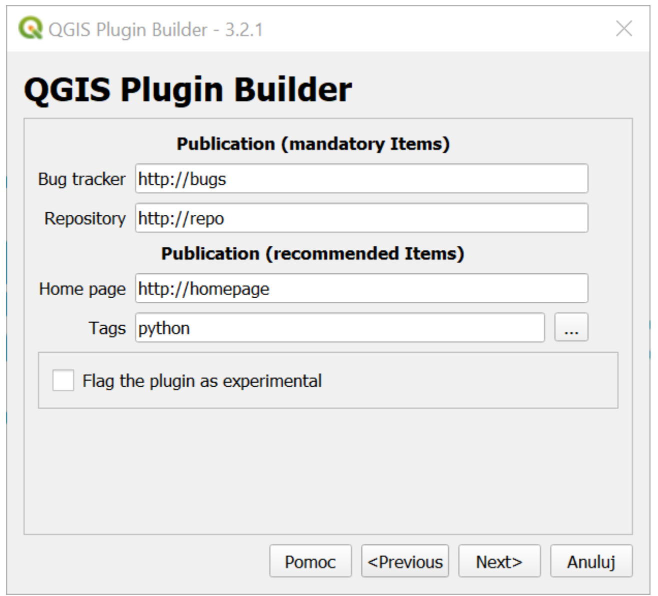


- **Internationalization** - dodaje katalog i pliki do tłumaczenia wtyczki na inne języki.
- **Help** - tworzy szablon do generowania pomocy HTML za pomocą narzędzia Sphinx.
- **Unit tests** - tworzy podstawowy zestaw testów dla wtyczki.
- **Helper scripts** - dodaje skrypty ułatwiające publikację wtyczki w oficjalnym repozytorium, tłumaczenie oraz testowanie.
- **Makefile** - dodaje Makefile pozwalający skompilować wtyczkę za pomocą GNU make.
- **pb_tool** - tworzy plik konfiguracyjny dla narzędzia pb_tool ułatwiającego m.in. kompilowanie, testowanie i tłumaczenie wtyczki.

Dodatkowe informacje

Są wymagane, jeśli wtyczka ma zostać opublikowana w oficjalnym repozytorium QGIS. W takim wypadku aby została ona zaakceptowana przez administratorów tego repozytorium kod wtyczki musi być publicznie dostępny (najczęściej korzysta się ze specjalnych serwisów

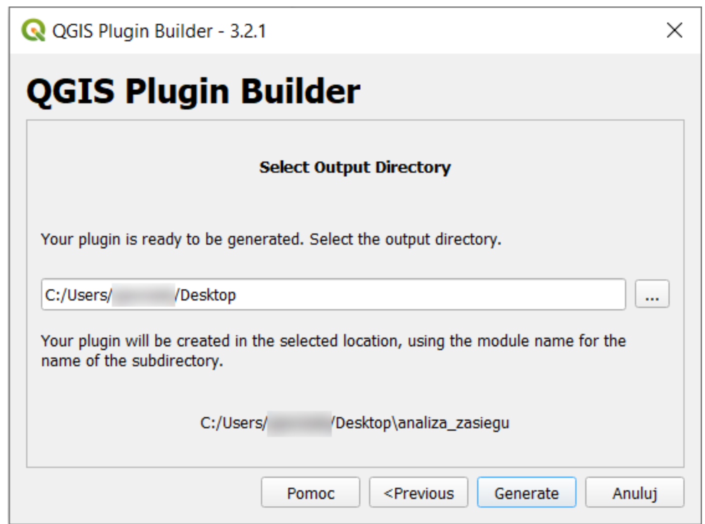
do tego przeznaczonych jak Github, Gitlab, Bitbucket czy Google Code) oraz musi istnieć jakiś sposób na kontakt z autorem w celu zgłoszenia uwag lub błędów. Dodatkowo możliwe jest podanie strony domowej z dokumentacją (jeśli nie jest ona dołączona do samej wtyczki).



- **Bug tracker** - adres serwisu, w którym można zgłaszać uwagi/błędy przez użytkowników.
- **Repository** - adres do kodu źródłowego wtyczki.
- **Home page** - strona domowa wtyczki.
- **Tags** - tagi opisujące funkcjonalność wtyczki. Wykorzystywane np. w oficjalnym repozytorium wtyczek.
- **Flag the plugin as experimental** - oznaczenie wtyczki jako eksperymentalnej.

Katalog wyjściowy

W ostatnim kroku należy wskazać katalog, gdzie zostanie utworzony nowy folder zawierający wszystkie pliki wtyczki. Na nasze potrzeby zapiszemy ją tymczasowo na pulpicie systemowym.



Struktura wtyczki

Utworzony szablon wtyczki składa się z kilku plików, ich liczba może być różna w zależności od wybranych dodatkowych elementów w kreatorze (np. pliki tłumaczeń). Jeśli nic nie zostało zaznaczone to lista powinna wyglądać w poniższy sposób:

__init__.py
analiza_zasiegu.py
analiza_zasiegu_dockwidget.py
analiza_zasiegu_dockwidget_base.ui
icon.png
metadata.txt
README.html
README.txt
resources.py
resources.qrc

- **metadata.txt**

Plik w formacie INI zawierający metadane wtyczki np. autora, nazwę, wersję. Większość informacji, które tu się znajdują została wygenerowana automatycznie na podstawie danych wprowadzonych w *Plugin Builder*.

- **resources.qrc i resources.py**

Pliki z zasobami *Qt*. Plik *resources.qrc* jest to format XML programu *Qt Designer* określający ścieżki do zasobów wtyczki np. ikon.

- **analiza_zasiegu_dockwidget_base.ui**

W plikach z rozszerzeniem .ui przechowywana jest definicja elementów graficznych np. okien dialogowych lub paneli dokowanych. Można je edytować za pomocą aplikacji *Qt Designer*, która jest opisana w części dotyczącej frameworka *Qt*.
Nazwa pliku uzależniona jest od nazwy modułu podanej w kreatorze.

- **__init__.py**

Plik jest generowany automatycznie i jest niezbędny do poprawnego uruchomienia wtyczki przez QGIS.
Funkcja `classFactory` służy do utworzenia głównej instancji głównej klasy wtyczki. Podczas tego procesu przekazywana jest instancja klasy `QgisInterface` (zmienna `iface`) za pomocą której wtyczka może komunikować z QGIS.

- **analiza_zasiegu.py**

Plik wtyczki, który jest importowany w *__init__.py* i zawiera główną klasę wtyczki. Odpowiada ona za całą obsługę wtyczki z poziomu QGIS. W niej m.in. mogą być tworzone i rejestrowane elementy graficzne jak panele dokowane lub okna dialogowe, które pojawiają się przy starcie wtyczki. Nazwa pliku uzależniona jest od nazwy modułu podanej w kreatorze.
Główna klasa wtyczki musi zawierać trzy metody wywoływane podczas ładowania lub wyłączania wtyczki:

- **`__init__`** - służy do stworzenia instancji klasy wtyczki w momencie jej uruchomienia, jako argument otrzymuje instancję klasy **`QgisInterface`**, dzięki której może komunikować się z instancją QGIS.
- **`initGui`** - jest wywoływana w momencie włączenia wtyczki i służy do dodawania elementów interfejsu (przyciski, menu, panele), konfiguracji oraz rejestracji sygnałów i slotów.
- **`unload`** - jest wywoływana podczas wyłączania wtyczki i pozwala usunąć elementy interfejsu oraz rozłączyć istniejące sygnały i sloty.

Ćwiczenie

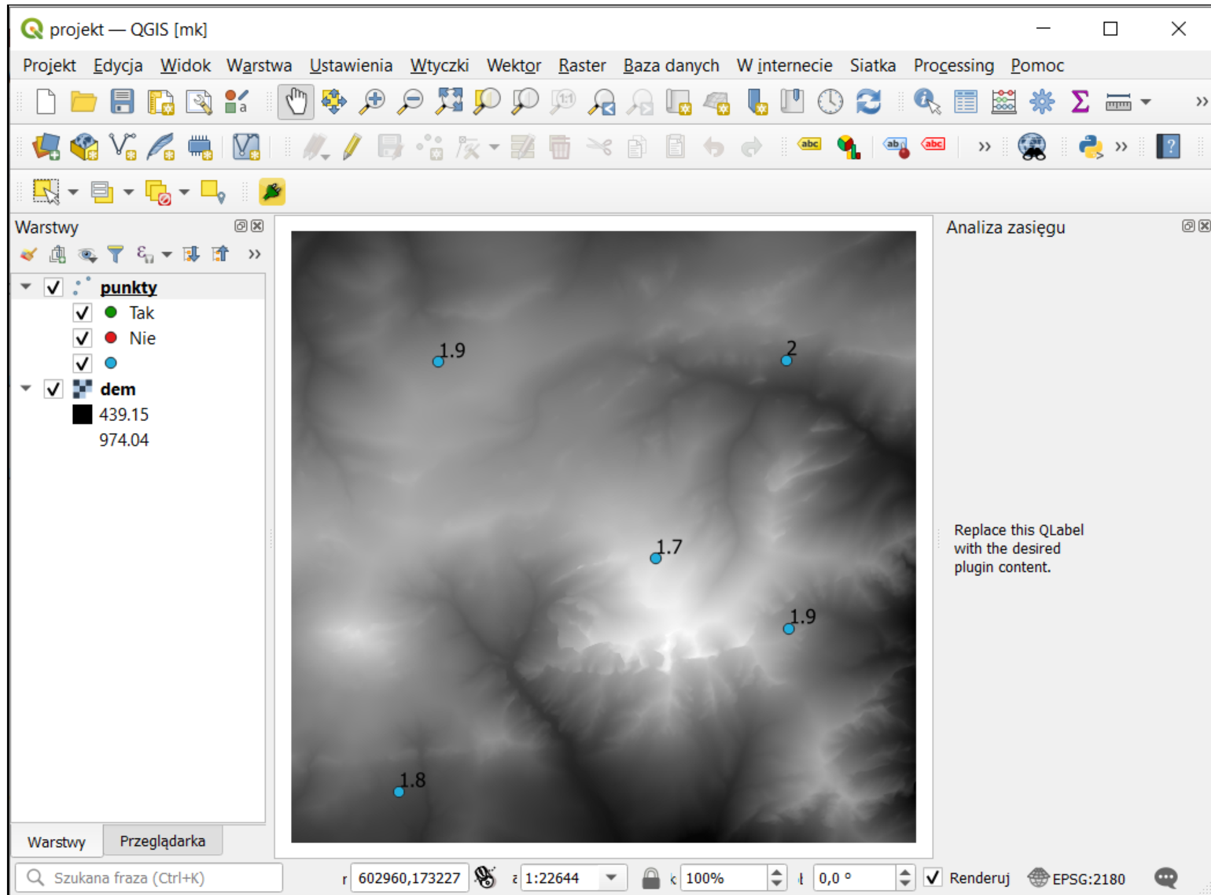
Treść zadania

Skopiuj katalog utworzonej wtyczki do folderu, w którym QGIS przechowuje zainstalowane wtyczki.

Opis

Katalog wtyczek można znaleźć wybierając w menu QGIS *Ustawienia -> Profile użytkownika -> Otwórz katalog aktywnego profilu* i przechodząc do katalogu *python/plugins*. Jeśli katalog *python* nie istnieje to można go utworzyć ręcznie. Następnie kopiujemy wtyczkę do tego folderu.

Po tym możliwe jest uruchomienie wtyczki, w tym celu należy zresetować QGIS (jeśli jest włączony), wejść w menu *Wtyczki -> Zarządzanie wtyczkami…*, zlokalizować wtyczkę *Analiza zasięgu* i ją włączyć. Na pasku narzędzi QGIS powinien pojawić się przycisk wtyczki i po jego kliknięciu zostanie wyświetlony panel boczny.



Plugin Reloader

Każda zmiana w kodzie wtyczki wymaga jej przeładowania. W tym celu należy ją włączyć i wyłączyć aby QGIS wczytał najnowszą wersję. Można to zrobić z poziomu *Menedżera wtyczek* lub resetując QGIS, ale jest to czasochłonne jeśli pracujemy nad kodem i co chwilę chcemy sprawdzić działanie po dokonanych zmianach.
Z pomocą przychodzi kolejna wtyczka dostępna w oficjalnym repozytorium o nazwie *Plugin Reloader*. Umożliwia ona na szybkie przeładowanie wskazanej wtyczki bez konieczności zamykania aplikacji lub włączenia *Menedżera wtyczek*.
Plugin Reloader dostępny jest w oficjalnym repozytorium QGIS. Przed przystąpieniem do dalszych ćwiczeń należy ją zainstalować za pomocą Menedżera wtyczek QGIS (menu *Wtyczki -> Zarządzanie wtyczkami*).

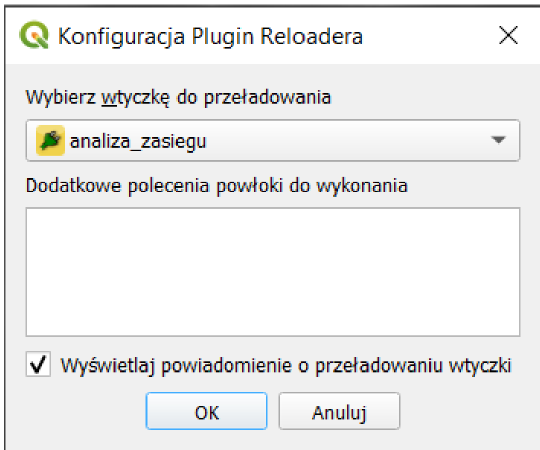
Ćwiczenie

Treść zadania

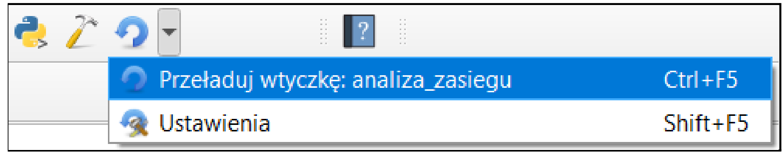
Skonfiguruj *Plugin Reloader* aby możliwe było resetowanie wtyczki *Analiza zasięgu*.

Opis

Należy kliknąć strzałkę znajdującą się z prawej strony ikony Plugin Reloader i wybrać *Ustawienia*. W otwartym oknie wskazujemy wtyczkę *analiza_zasiegu* i klikamy *OK*.



Po tym należy kliknąć przycisk i sprawdzić czy wtyczka została poprawnie zresetowana.



Framework Qt

Qt to pakiet bibliotek i narzędzi, służących głównie tworzeniu wieloplatformowych graficznych interfejsów aplikacji (GUI). Qt jest dostępne na wszystkich głównych systemach operacyjnych.

Biblioteki Qt dostępne są dla C++, a dzięki nakładkom można korzystać z ich możliwości w wielu innych językach programowania, w tym w Python. Służy do tego bibliotek *PyQt*.

Import z głównych modułów:

```
# Główny moduł z elementami niegraficznymi, mechanizmem sygnałów i slotów, wyrażeniami regularnymi itp.
from qgis.PyQt.QtCore import *

# Klasy graficzne służące do tworzenia okien dialogowych, obsługi kolorów i czcionek, rysowania 2D i 3D itp.
from qgis.PyQt.QtGui import *

# Zbiór kontrolek graficznych tzw. widżetów
from qgis.PyQt.QtWidgets import *
```

Zasoby

Zasoby w Qt służą głównie do przechowywania plików graficznych np. ikon. Informacje o zasobach są przechowywane w plikach .qrc. Jest to format oparty na XML i ma strukturę:

```
<RCC>
    <qresource prefix="prefiks" >
        <file>plik.png</file>
    </qresource>
</RCC>
```

Dodając nowe zasoby należy skopiować fragment element **`qresource`**. W QGIS jako prefiks podaje **`/plugins/<nazwa_wtyczki>`**.

Python natywnie nie wspiera tego formatu i w celu skorzystania z niego wymagane jest skompilowanie go do pliku .py po każdej zmianie. Służy do tego polecenie **`pyrcc5`** z poziomu konsoli *OSGeo4W Shell*:

```
pyrcc5 -o resources.py resources.qrc
```

Za opcją `-o` (output) należy podać nazwę pliku .py, który zostanie utworzony, jeśli plik istnieje to zostanie on nadpisany. Najbezpieczniej podać pełne ścieżki do obu plików.

W przypadku gdy po wpisaniu polecenia **pyrcc5** pojawi się informacja, że jest ono nieznane należy dodatkowo wpisać polecenie **qt5_env**, aby odpowiednio skonfigurować konsolę. Dotyczy to jednak tylko starszych instalacji QGIS.
Aby wskazać zasób w kodzie Pythona należy podać prefiks wraz z nazwą zasobu poprzedzone dwukropkiem:

```
":/plugins/nazwa_wtyczki/nazwa_zasobu"
```

Ćwiczenie

Treść zadania

Zmień ikonę wtyczki na plik *icon.png* z katalogu ćwiczeniowego .

```
git checkout v_2
```

Opis
Należy skopiować plik graficzny i nadpisać istniejący o tej samej nazwie w katalogu wtyczki. Następnie uruchamiamy konsolę OSGeo4w Shell i wpisujemy polecenie:

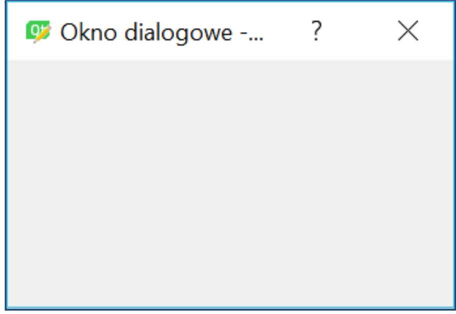
```
pyrcc5 -o C:\Users\...\plugins\analiza_zasiegu\resources.py
C:\Users\...\plugins\analiza_zasiegu\resources.qrc
```

W celu ułatwienia wpisywania pełnych ścieżek pliki można przeciągnąć na okno konsoli, w miejsce kursora zostanie wstawiona pełna ścieżka do tego pliku.

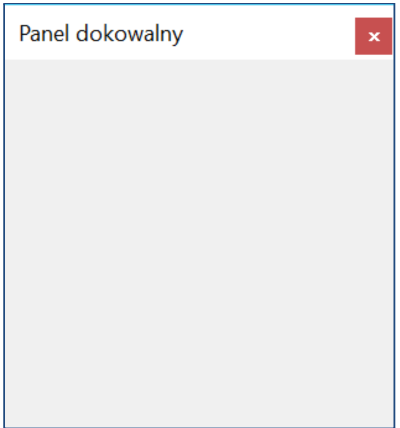
Widżety

Widżet (widget, kontrolka) – podstawowy element graficznego interfejsu użytkownika (np. okno, pole edycji, suwak, przycisk). Qt posiada bogaty zbiór podstawowych widżetów, z których można składać okna dialogowe.

- **QDialog** – okno dialogowe, swobodne okno, które użytkownik może przesuwać, może ono być zawsze na wierzchu okna aplikacji oraz je zablokować (tryb modalny).



- **QDockWidget** - panel, który może zostać “przyklejony” do jednej z krawędzi okna głównego, panele można również dokować jeden na drugim.



- **`QPushButton`** – przycisk

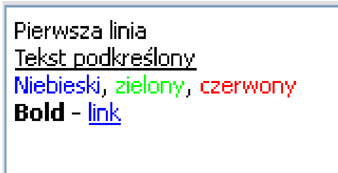
- **`QLabel`** – etykieta tekstowa

Text Label

- **`QLineEdit`** – okienko do wpisywania tekstu (jednoliniowe)

Enter your name

- **`QTextEdit`** – okienko do wpisywania tekstu (wieloliniowe)



- **`QSpinBox`** i **`QDoubleSpinBox`** - kontrolki do wprowadzania wartości liczbowych (całkowitych i dziesiętnych)

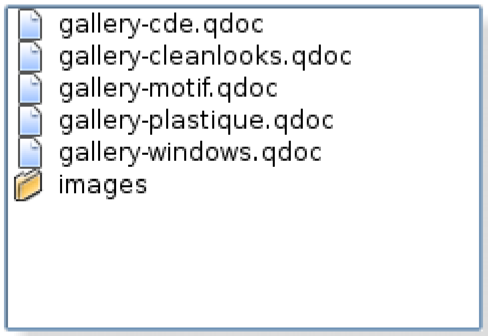
- **`QRadioButton`** – przycisk opcji

Search from the cursor

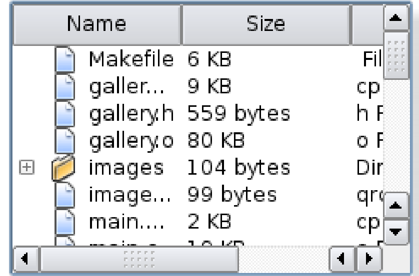
- **`QCheckBox`** – przycisk wyboru

- **`QComboBox`** – lista wyboru

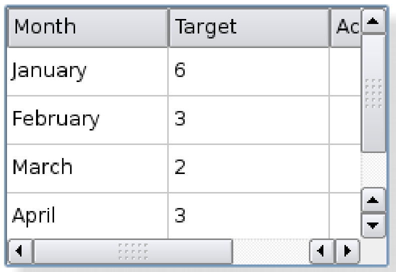
- **`QListWidget`** – lista elementów
- **`QListWidgetItem`** – element listy



- **`QTreeWidget`** – lista elementów z widokiem drzewa
- **`QTreeWidgetItem`** – element listy z widokiem drzewa



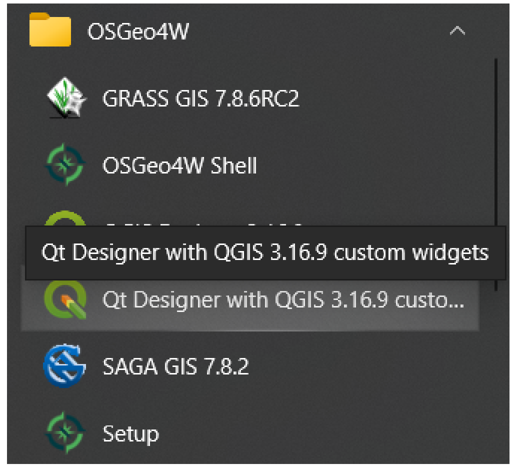
- **`QTableWidget`** – tabela
- **`QTableWidgetItem`** – element tabeli (komórka)



Qt Designer umożliwia również rejestrowanie dodatkowych widżetów. W ten sposób można również dodać wiele kontrolek pochodzących z QGIS.

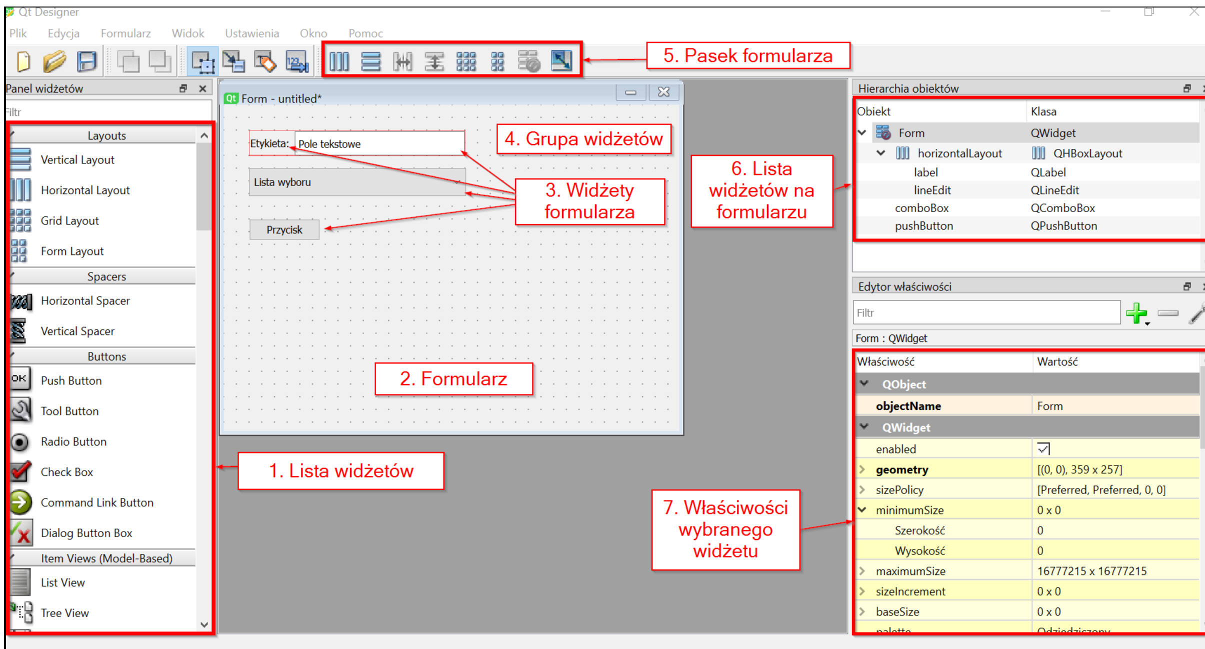
Qt Designer

Program *Qt Designer* pozwala wizualnie tworzyć okna dialogowe wykorzystywane przez pakiet *Qt*. Użytkownik może konfigurować wygląd formularza umieszczając na nim kontrolki. Do ich poprawnego rozmieszczenia służy system układów (*layouts*), dzięki którym mogą one się dostosowywać do zmiany rozmiaru okna. Z poziomu *Qt Designer* możliwe jest również ustalenie wielu właściwości kontrolek np. wyświetlane teksty, elementy list, czcionki, podpowiedzi, ramki i wiele więcej.
Okna dialogowe stworzone w programie *Qt Designer* zapisywane są w formacie UI (XML).
Aplikacja jest instalowana razem z QGIS i jest dostępna w menu *Start*.



Po uruchomieniu pojawia się okno umożliwiające utworzenie nowego pustego formularza lub wybrania jednego z szablonów.

Elementy interfejsu aplikacji Qt Designer



1. Lista widżetów - lista zawierająca graficzne kontrolki, które mogą zostać umieszczone na formularzu. Aby dodać widżet do formularza należy je na niego przeciągnąć.
2. Formularz - wizualna reprezentacja tworzonego okna, na którym rozmieszczone są widżety.
3. Widżety formularza - kontrolki na formularzu.
4. Grupa widżetów - kontrolki można grupować w różnych układach, jest to wizualizowane za pomocą czerwonej siatki.
5. Pasek formularza - przyciski do definiowania układów (*layouts*) dla widżetów lub formularza.
6. Lista widżetów na formularzu - lista widżetów w formie hierarchicznej, pokazuje jak kontrolki są pogrupowane.

7. Właściwości wybranego widżetu - atrybuty zaznaczonej kontrolki, które można definiować z poziomu edytora. Są one podzielone wg klas, z których dziedziczy dany widżet.

Układy (layouts)

Do sterowania rozmieszczeniem widżetów służy system układów (*layouts*). Rozmieszczają one widżety w siatce i to ona steruje ich ułożeniem i rozmiarem w stosunku do innych kontrolek lub całego formularza. Na pasku narzędzi dostępnych jest kilka przycisków do definiowania układów:

- *Rozmieść w poziomie* - rozmieszcza widżety w jednym wierszu,
- *Rozmieść w pionie* - rozmieszcza widżety w kolumnie,
- *Rozmieść poziomo w splitterze* - jak *Rozmieść w poziomie*, ale dodaje pomiędzy nimi pasek do zmiany szerokości,
- *Rozmieść pionowo w splitterze* - jak *Rozmieść w pionie*, ale dodaje pomiędzy nimi pasek do zmiany wysokości,
- *Rozmieść w siatce* - tworzy regularną siatkę, w której kontrolki mogą być rozmieszczone w kolumnach i wierszach,
- *Rozmieść w formularzu* - widżety ułożone są w dwóch kolumnach, lewa zawiera etykiety, prawa kontrolki edycyjne,
- *Usuń rozmieszczenie* - pozwala usunąć wskazany układ z formularza.

Układy działają w dwóch trybach. Jeśli zaznaczone są przynajmniej dwa widżety to po wybraniu rozmieszczenia są one grupowane i traktowane jako pojedyncza kontrolka. Drugi tryb jest aktywny jeśli żaden widżet nie jest zaznaczony (aktywny jest wtedy formularz) i wybranie układu spowoduje automatyczne rozmieszczenie wszystkich kontrolek w formularzu. Takie rozmieszczenie będzie również powodować, że przy zmianie rozmiaru okna wszystkie kontrolki będą dopasowywane dynamicznie wg wybranego układu.

Pliki .ui i Python

Pliki UI nie są natywnie wspierane przez Pythona. Aby z nich skorzystać w aplikacjach napisanych z pomocą PyQt należy je skompilować do plików .py narzędziem *pyuic5* lub bezpośrednio wczytać za pomocą funkcji **`loadUiType`**.

- wygenerowanie pliku .py w konsoli *OSGeo Shell*:

```
pyuic5 -o dialog.py dialog.ui
```

Po kompilacji w utworzonym pliku znajduje się klasa Python definiująca całe okno. Należy tą klasę zaimportować i użyć przy definiowaniu nowej klasy:

```
from qgis.PyQt.QtWidgets import QDockWidget
# Import klasy wygenerowanej przy kompilacji
from .dialog import KlasaOkna
```

```python
class PluginDockWidget(QDockWidget, KlasaOkna):
    def __init__(self, parent=None):
        # Trzeba wywołać odpowiednie funkcje w celu utworzenia okna
        super(MKSzkolenieDockWidget, self).__init__(parent)
        self.setupUi(self)
```

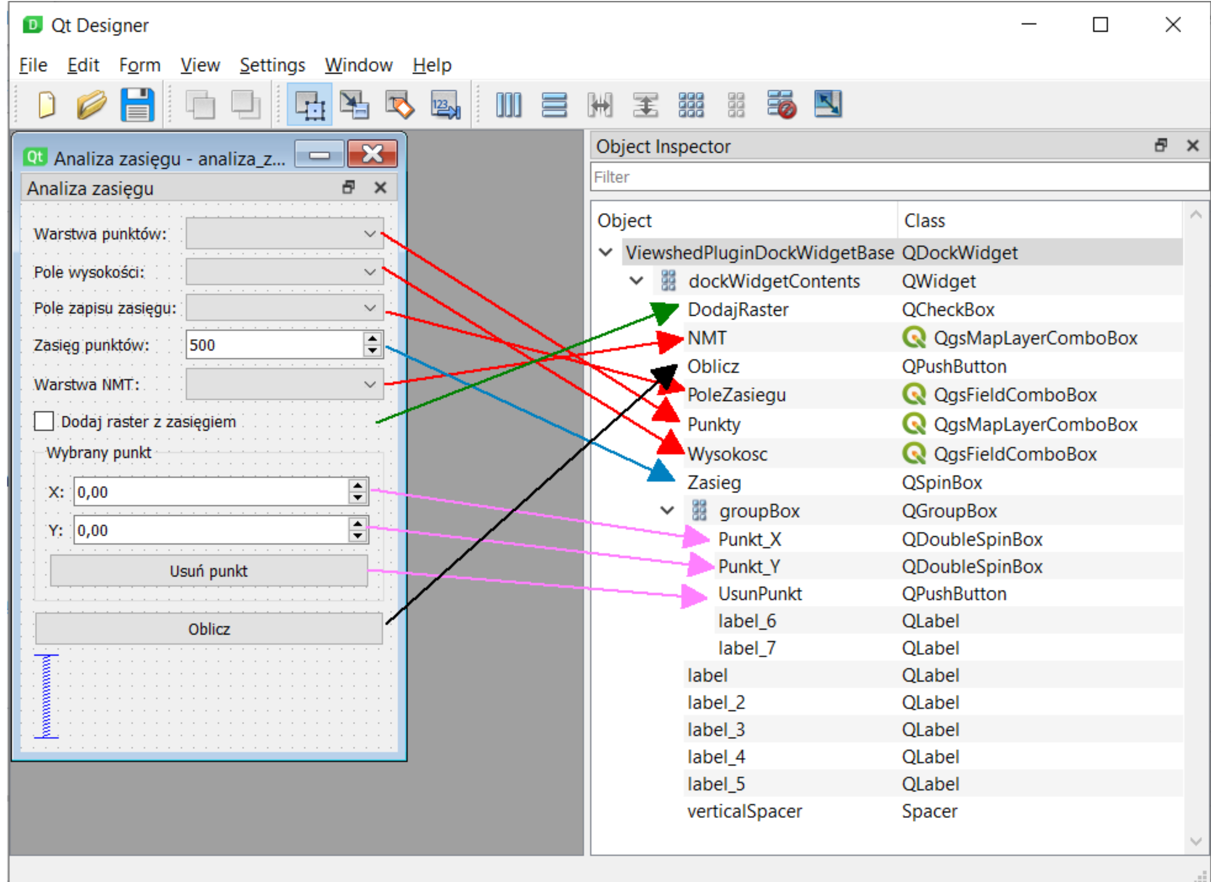
- dynamiczne ładowanie pliku .ui bez konieczności ręcznej kompilacji:

```python
# uic służy do dynamicznej kompilacji plików .ui do klas Pythona
from qgis.PyQt import uic
from qgis.PyQt.QtWidgets import QDockWidget
import os
# Kompilacja "w locie"
FORM_CLASS, _ = uic.loadUiType(os.path.join(
        os.path.dirname(__file__), 'plugin_dockwidget.ui' ))
# Użycie stworzonego obiektu jako klasy bazowej
class PluginDockWidget(QDockWidget, FORM_CLASS):
    ...
```

Ten sposób jest wykorzystywany przez *Plugin Builder* w wygenerowanych szablonach.

Ćwiczenie

Treść zadania
Za pomocą aplikacji *Qt Designer* stwórz okno wtyczki wg poniższego wzoru.

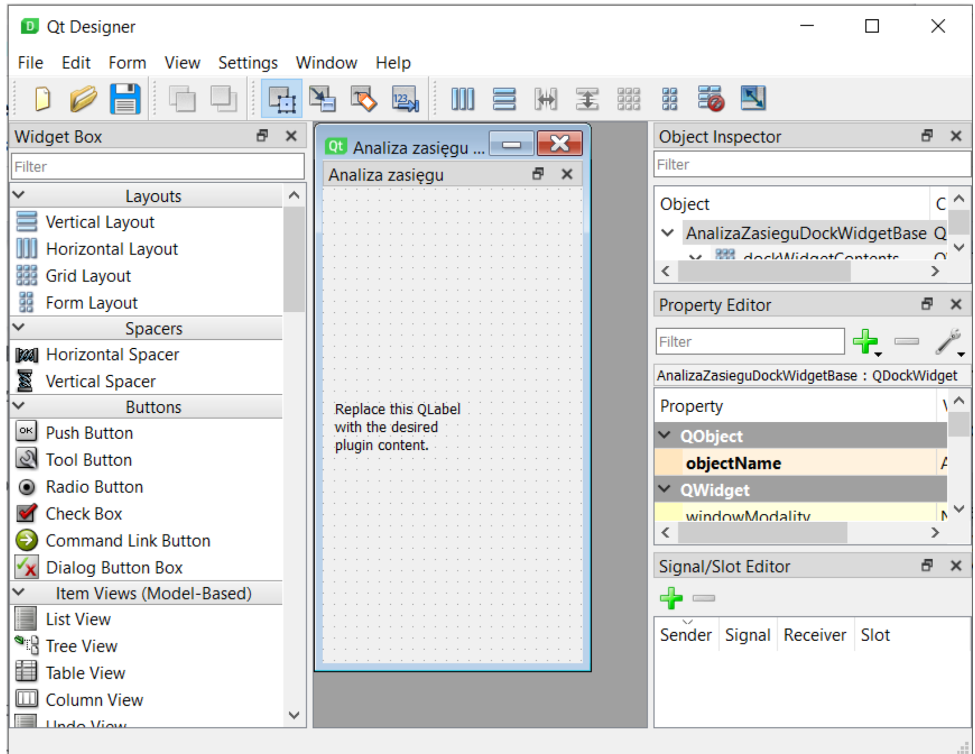


Do określenia układu należy użyć opcji *Rozmieść w formularzu*.
W edytorze właściwości kontrolek ustaw wartości zgodnie z listą:

- *Zasieg*:
    - wartość maksymalna: 10 000
    - wartość wyświetlana: 500
- *Punkt_X* i *Punkt_Y*:
    - wartość maksymalna: 9 999 999 999
    - tylko do odczytu: tak

```
git checkout v_3
```

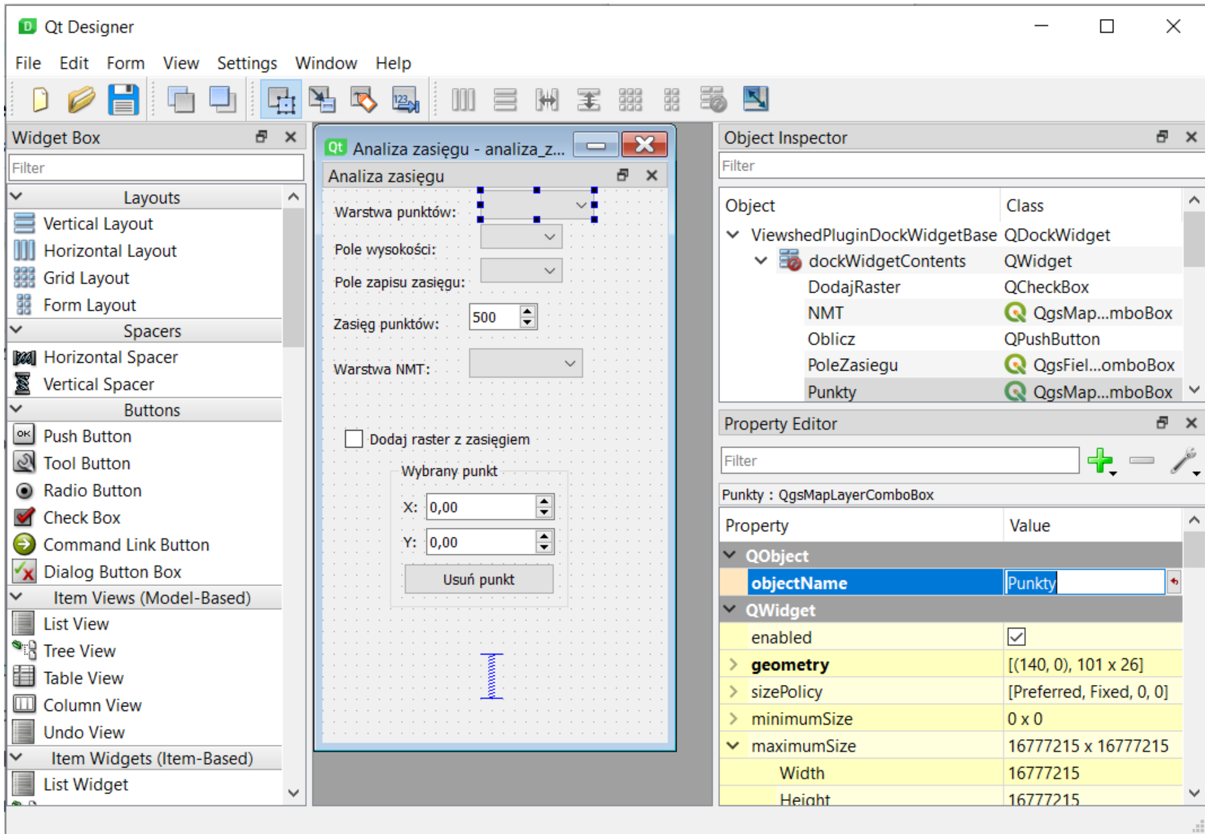
Opis
Po uruchomieniu aplikacji *Designer* pojawi się okno do stworzenia nowego formularza. Można z jego poziomu wybrać opcję *Open* i wskazać plik *analiza_zasiegu_dockwidget_base.ui* z katalogu wtyczki lub je zamknąć i przeciągnąć plik na okno aplikacji (jeśli katalog wtyczki jest otworzony w Eksploratorze).



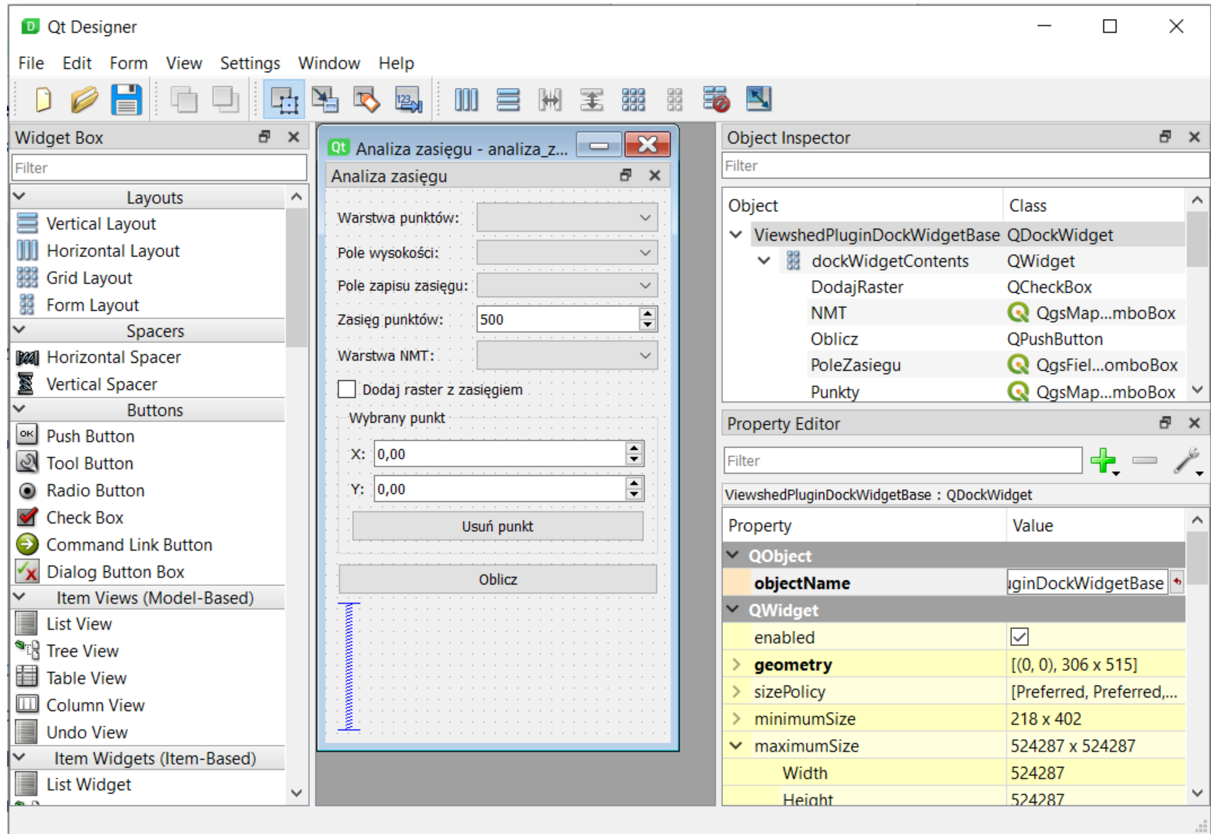
Na początku należy wyczyścić istniejące elementy. W tym celu klikamy przycisk usunięcie rozmieszczenia (przycisk), zaznaczamy istniejącą etykietę (***`QLabel`***) i ją usuwamy (klawisz *Del*).

Następnie na formularzu należy umieścić wskazane kontrolki wg typów określonych na liście. Kontrolki należy umieścić mniej więcej tak jak mają się ostatecznie znaleźć w oknie.

Każda kontrolka edycyjna powinna mieć nadaną nazwę (również wg listy z obrazka), w tym celu należy zaznaczyć dany widżet i zmodyfikować w *Edytorze właściwości* atrybut ***`objectName`*** ustawiając podane nazwy.

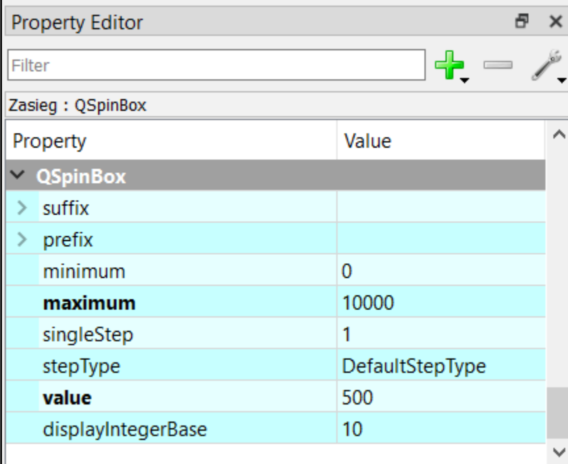


Po zakończeniu należy w pasku narzędzi wybrać opcję *Rozmieść w formularzu* . Jeśli efekt będzie odbiegał od pożądanego można spróbować poprawić wygląd przesuwając widżety albo cofnąć rozmieszczenie (przycisk), poprawić umiejscowienie kontrolek i ponowić rozmieszczenie w formularzu.

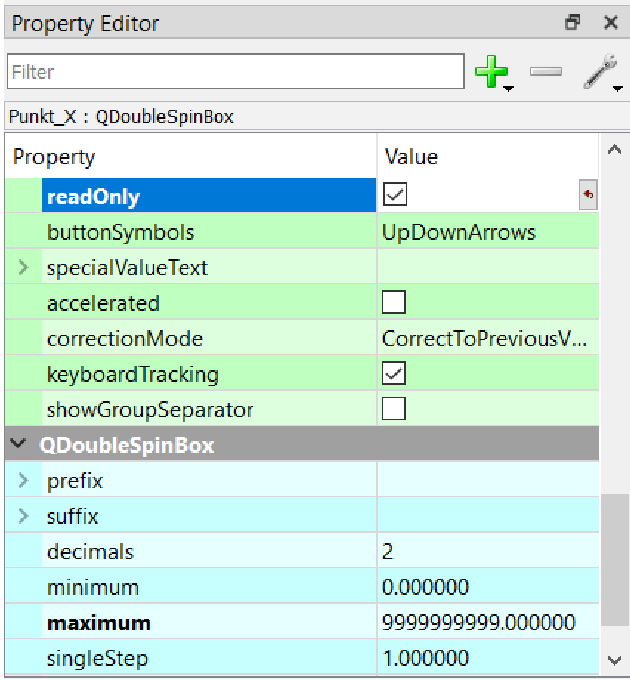


Aby ustawić właściwości kontrolek należy je zaznaczyć i w *Edytorze właściwości* odszukać parametry:

- *Zasieg*:
  - wartość maksymalna: ustawić parametr **`maximum`** na *10000*
  - wartość wyświetlana: ustawić parametr **`value`** na *500*



- *Punkt_X* i *Punkt_Y*:
  - wartość maksymalna: ustawić parametr **`maximum`** na *9999999999*
  - tylko do odczytu: zaznaczyć parametr **`readOnly`**



Sygnały i sloty

Dzięki sygnałom i slotom możliwe jest ustalenie sposobu przekazywania informacji pomiędzy elementami aplikacji. Sygnał emitowany jest w przypadku wystąpienia danej akcji np. wciśnięcie przycisku. Aplikacja po wystąpieniu sygnału wykonuje funkcję (tzw. slot), z którą sygnał został wcześniej połączony.

- sygnał może być połączony z wieloma slotami
- sygnał może być połączony z innym sygnałem
- slot może być połączony z wieloma sygnałami
- sygnały mogą być emitowane „ręcznie” za pomocą metody emit()

Połączenie sygnału ze slotem

```
obiekt.sygnał.connect( slot )
```

Rozłączenie sygnału ze slotem

```
obiekt.sygnał.disconnect( slot )
```

Sygnał może przekazywać argumenty. W takim wypadku można określić je przy łączeniu ze slotem. Jest to opcjonalne, jeśli istnieje jedna wersja sygnału, jeśli występuje on w kilku wersjach (różne argumenty) trzeba określić, która wersja nas interesuje.

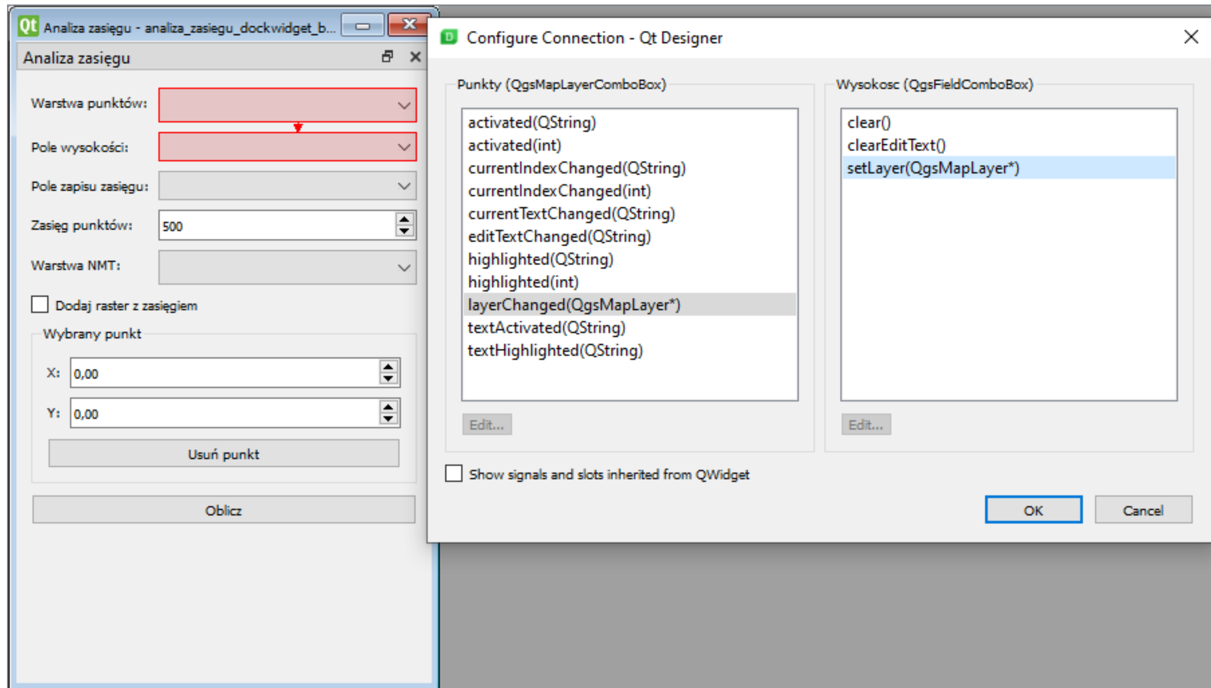
```
# Sygnał wysyła liczbę całkowitą
obiekt.sygnał[int].connect(slot)

# Sygnał wyśle obiekt typu QgsFeature
```

```
obiekt.sygnał[QgsFeature].connect(slot)

# Sygnał wyśle dwa argumenty, pierwszy liczbę, drugi słownik
obiekt.sygnał[int, dict].connect(slot)
```

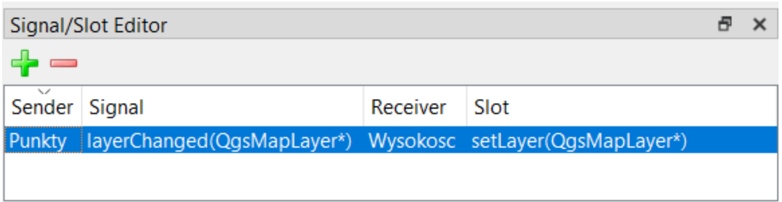
Niektóre sygnały i sloty można również ustawić bezpośrednio w aplikacji *Qt Designer*. Dzięki temu można bez konieczności pisania kodu powiązać ze sobą dwa elementy i wywołać akcje. Przykładem może być zamknięcie okna dialogowego po kliknięciu przycisku lub odświeżenie danych po zmianie wartości w polu wyboru. Służy do tego drugi widok, który można uruchomić wybierając z menu *Edit* pozycję *Edit Signals/Slots* lub klikając na pasku narzędzi przycisk .

Teraz można łączyć ze sobą po dwie kontrolki. w tym celu należy kliknąć i przytrzymać lewy przycisk myszy na widżecie, który ma wysyłać sygnał. Następnie najechać na widżet, który otrzyma informację o wyemitowaniu sygnału. Połączenie między kontrolkami będzie wizualizowane strzałką.

Po puszczeniu klawisza myszy pojawi się nowe okno. Po lewej stronie znajduje się lista sygnałów z pierwszej kontrolki, a po prawej lista slotów drugiej kontrolki. Jeśli na liście nie ma szukanych elementów można zaznaczyć opcję *Show signals and slots inherited from QWidget*, dzięki czemu wyświetlone zostaną dodatkowe pozycje, które są dziedziczone z klasy **`QWidget`**. Jest to klasa bazowa dla wszystkich widżetów w Qt.



Należy wskazać jeden sygnał i jeden slot po czym kliknąć OK. Na liście *Signal/Slot Editor* pojawi się nowa pozycja.



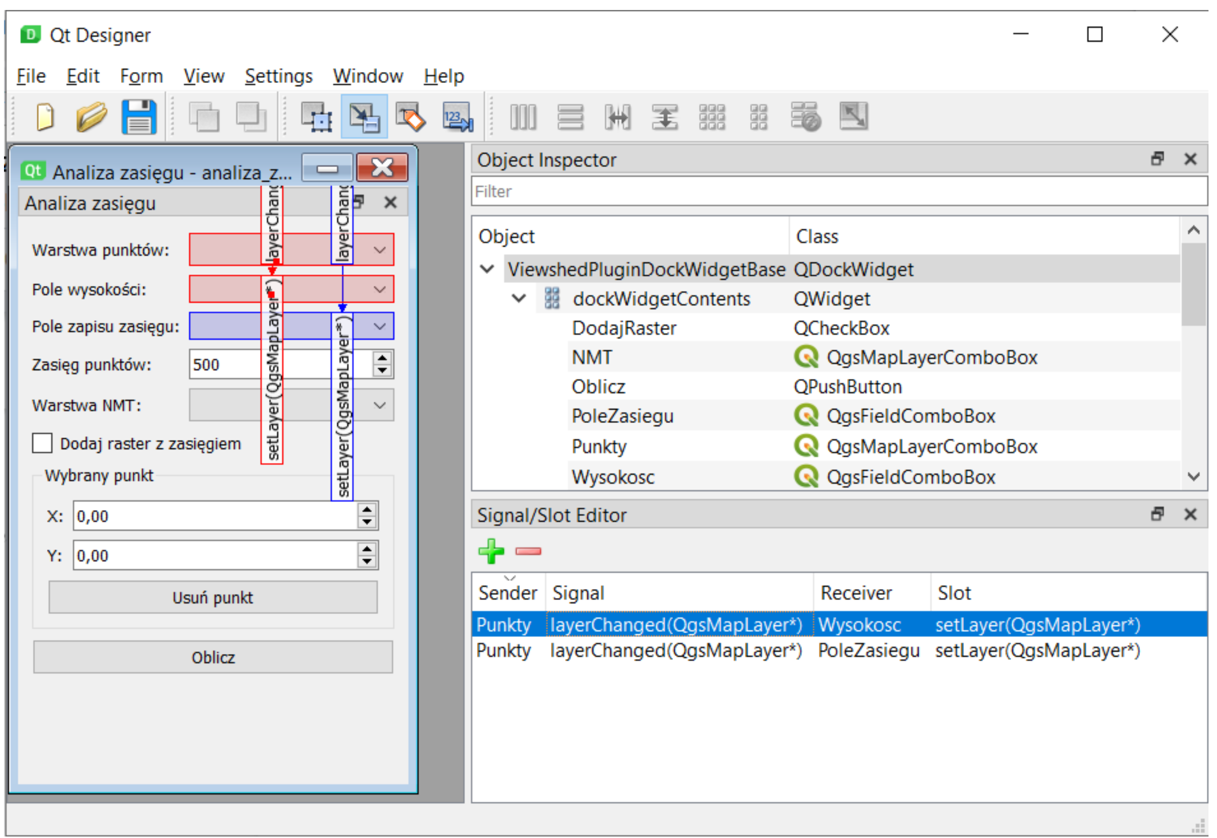
Kolejne sygnały możemy dodawać w analogiczny sposób. Można również połączyć dwie kontrolki kilka razy.
Aby powrócić do standardowego widoku edycji formularza należy z menu *Edit* wybrać opcję *Edit Widgets* lub na pasku narzędzi kliknąć przycisk .

Ćwiczenie

Treść zadania
Używając widoku edytora sygnałów w *Qt Designer* połącz zmianę warstwy punktowej (sygnał *layerChanged*) z aktualizacją pola wysokości i zapisu zasięgu (slot *setLayer*).

```
git checkout v_4
```

Opis
Należy włączyć edytor sygnałów *Qt Designer* wybierając z menu *Edit* pozycję *Edit Signals/Slots* lub klikając na pasku narzędzi przycisk . Aby wskazać relację między dwiema kontrolkami należy najechać kursorem myszy nad tą, która będzie wysyłała sygnał. W naszym przypadku jest to widżet *Punkty*. Następnie klikamy i trzymając wciśnięty lewy klawisz myszy najeżdżamy na drugą kontrolkę - tą która ma zostać poinformowana o wysłanym sygnale. Pojawi się strzałka informująca o kierunku przepływu informacji i pojawi się okno dialogowe do zdefiniowania sygnału i slotu.
Z lewej strony należy wskazać sygnał, na który będziemy reagować, w tym przypadku interesuje na zmiana warstwy przez użytkownika czyli **`layerChanged`**. Zaznaczamy tą pozycję. W nawiasie podana jest informacja jakie dane są wysyłane razem z sygnałem (**`QgsMapLayer`** - warstwa QGIS).
Z prawej strony musimy wskazać slot kontrolki **`Wysokosc`**, który zostanie uruchomiony. Metoda **`setLayer`** klasy **`QgsFieldComboBox`** pozwala ustawić warstwę, z której pobrana zostanie lista pól. Jak widać przyjmuje ona ten sam typ danych co wysyłany sygnał - **`QgsMapLayer`**. Oznacza to, że możemy połączyć oba elementy i wskazana warstwa trafi jako argument metody **`setLayer`**.
Analogicznie postępujemy dla kontrolki **`PoleZasiegu`**.



Qt Designer
File Edit Form View Settings Window Help
Analiza zasięgu - analiza_z...
Analiza zasięgu
Warstwa punktów:
Pole wysokości:
Pole zapisu zasięgu:
Zasięg punktów:
500
Warstwa NMT:
Dodaj raster z zasięgiem
Wybrany punkt
X:
0,00
Y:
0,00
Usuń punkt
Oblicz
setLayer(QgsMapLayer*)
Object Inspector
Filter
Object
Class
ViewshedPluginDockWidgetBase
QDockWidget
dockWidgetContents
QWidget
DodajRaster
QCheckBox
NMT
QgsMapLayerComboBox
Oblicz
QPushButton
PoleZasiegu
QgsFieldComboBox
Punkty
QgsMapLayerComboBox
Wysokosc
QgsFieldComboBox
Signal/Slot Editor
Sender
Signal
Receiver
Slot
Punkty
layerChanged(QgsMapLayer*)
Wysokosc
setLayer(QgsMapLayer*)
Punkty
layerChanged(QgsMapLayer*)
PoleZasiegu
setLayer(QgsMapLayer*)

Rozwijanie wtyczki

Dostęp do widżetów

Okna dialogowe są zdefiniowane jako klasy. Dziedziczą one zazwyczaj dwie klasy. Pierwsza określa rodzaj okna dialogowego *Qt*. W zależności od wyboru rodzaju okna podczas generowania wtyczki z szablonu może to być klasa **`QDialog`** (okno dialogowe) lub **`QDockWidget`** (panel dokowany). Są tu zdefiniowane podstawowe zachowania danego rodzaju okna np. obsługę przyciągania w panelach do krawędzi okna głównego aplikacji.
Druga dziedziczona klasa (*Plugin Builder* nazywa ją **`FORM_CLASS`**) zawiera wszystkie informacje o użytych widżetach i ich rozmieszczeniu w formularzu. Jest to skompilowany do kodu Python plik UI, który został przygotowany w aplikacji *Designer*. Z punktu widzenia programisty najważniejszym elementem tej klasy jest metoda **`setupUi`**. Po jej wywołaniu w metodzie **`__init__`** klasy okna mamy dostęp do wszystkich widżetów, które zostały utworzone w *Qt Designer*. Są one zdefiniowane jako atrybuty klasy więc wystarczy podać ich nazwę (atrybut **`objectName`** w *Qt Designer*) po obiekcie **`self`**:

```
# Kompilacja okna z formatu UI do obiektu Python
FORM_CLASS, _ = uic.loadUiType(os.path.join(
    os.path.dirname(__file__), 'okno.ui'))

class KlasaOkna(QtWidgets.QDockWidget, FORM_CLASS):
    def __init__(self):
        ...
        self.setupUi()
        # Po wywołaniu setupUi mamy dostęp do zdefiniowanych kontrolek
        print( self.widzet )
```

Każdy widżet jest instancją konkretnej klasy, a co za tym idzie posiada zestaw metod, atrybutów i sygnałów. Np. kontrolki edycyjne mają metody, które pozwalają z poziomu kodu ustawić ich wartości (np. tekst), jak również je zwrócić. Najczęściej metody do ustawiania wartości rozpoczynają się od przedrostka **`set`**. I tak np. metoda widżetu **`setText( str )`** może ustawić tekst, a **`text()`** zwrócić wyświetlaną wartość.

```
# Ustawienie wyświetlanego tekstu
pole_tekstowe.setText( 'Tekst do wyświetlenia' )
# Zwrócenie tekstu i przypisanie do zmiennej
tekst = pole_tekstowe.text()
print( tekst )
# Tekst do wyświetlenia
```

Wszystkie metody i sygnały danego typu widżetu są opisane w jego dokumentacji.

QPushButton

Przycisk, który reaguje na kliknięcie kursorem myszy.

- **`clicked`** - sygnał wysyłany po kliknięciu przycisku. Można się z nim połączyć w celu wywołania określonej akcji.

QCheckBox

Przycisk wyboru, którym można określać włączenie/wyłączenie danej opcji w aplikacji.

- **`setChecked( bool )`**, **`isChecked()`** - ustawienie i zwrócenie informacji czy pole jest zaznaczone.

QSpinBox i QDoubleSpinBox

Pole edycyjne pozwalające na wprowadzanie przez użytkownika wartości numerycznych. **`QSpinBox`** pozwala wyświetlać liczby całkowite (**`int`**), a **`QDoubleSpinBox`** zmiennoprzecinkowe (**`float`**).

- **`setValue( int/float )`**, **`value()`** - ustawienie i zwrócenie wyświetlanej wartości liczbowej.
- **`setMinimum( int/float )`**, **`setMaximum( int/float )`**, **`minimum()`**, **`maximum()`** - metody ustawiające i zwracające minimalne i maksymalne wartości możliwe do wprowadzenia.
- **`setDecimals( int )`**, **`decimals()`** - dotyczy **`QDoubleSpinBox`**, ustawienie i zwrócenie dokładności liczb rzeczywistych (wyświetlanych miejsc po przecinku).
- **`setSpecialValueText( str )`**, **`specialValueText()`** - ustawienie specjalnego tekstu, który jest wyświetlany gdy wartość jest równa **`minimum()`**.

QgsMapLayerComboBox

Pole wyboru QGIS do wyświetlania wczytanych warstw. Dostępna lista jest automatycznie aktualizowana w momencie dodawania/usuwanie warstw. Pozwala m.in. filtrować listę pod kątem konkretnego typu warstwy.

- **`setLayer( QgsMapLayer )`**, **`currentLayer()`** - ustawia i zwraca wybraną warstwę.
- **`setFilters( QgsMapLayerProxyModel.Filters )`**, **`filters()`** - ustawia i zwraca filtr warstw.
- **`setShowCrs( bool )`**, **`showCrs()`** - ustawia i zwraca informację, czy przy warstwach ma się wyświetlać ich układ współrzędnych.

QgsFieldComboBox

Pole wyboru QGIS do wyświetlania listy pól określonej warstwy wektorowej. Możliwe jest filtrowanie pól wg typu.

- **`setLayer( QgsMapLayer )`**, **`layer()`** - ustawia i zwraca wybraną warstwę, dla której mają być wyświetlone pola.

- **setFilters(QgsFieldProxyModel.Filters), filters()** - ustawia i zwraca filtr typu pól.
- **setField(str), currentField()** - ustawia i zwraca nazwę aktualnie wybranego pola.

Ćwiczenie

Treść zadania

Wykorzystując dokumentację QGIS API ustaw dla list rozwijanych filtry:

- NMT - wyświetlanie warstw rastrowych,
- Punkty - wyświetlanie warstw punktowych,
- Wysokosc - wyświetlanie pól numerycznych,
- PoleZasiegu - wyświetlanie pól tekstowych.

```
git checkout v_5
```

Opis

Do ustawienia filtra dla kontrolki **QgsMapLayerComboBox** służy metoda **setFilters**. Zgodnie z dokumentacją jako argument należy przekazać atrybut klasy **QgsMapLayerProxyModel**, w przypadku warstw rastrowych jest to **RasterLayer**, a punktowych **PointLayer**.

Do ustawienia filtrów widżetu **QgsFieldComboBox** również służy metoda **setFilters**, jednak jako argument podajemy odpowiedni atrybut klasy **QgsFieldProxyModel**. Dla pól numerycznych ustawiamy **Numeric**, a dla tekstowych **String**.

Kod źródłowy

Plik *analiza_zasiegu_dockwidget.py*

```
# Import klas QGIS API
from qgis.core import QgsMapLayerProxyModel, QgsFieldProxyModel

class AnalizaZasieguDockWidget(QtWidgets.QDockWidget, FORM_CLASS):

    def __init__(self, parent=None):
        self.setupUi(self)

        self.NMT.setFilters( QgsMapLayerProxyModel.RasterLayer )
        self.Punkty.setFilters( QgsMapLayerProxyModel.PointLayer )
```

```
        self.Wysokosc.setFilters( QgsFieldProxyModel.Numeric )
        self.Wysokosc.setField( 'Wysokosc' )

        self.PoleZasiegu.setFilters( QgsFieldProxyModel.String )
        self.PoleZasiegu.setField( 'W_zasiegu' )
```

Akcje

Akcje (**`QAction`**) są to polecenia, które mogą być wywołane z poziomu paska narzędzi, menu lub skrótów klawiaturowych. Aby dodać nową akcję wtyczki należy skorzystać z metody **`add_action`**. Metoda ta pozwala zarejestrować nową akcję, jako jej argumenty podaje się różne opcje konfiguracyjne np. ikonę, wyświetlany tekst oraz funkcję, która ma być uruchomiona w momencie wywołania danej akcji.

```
    def add_action(
        self,
        icon_path,              # Ścieżka do pliku ikony
        text,                   # Wyświetlany tekst
        callback,               # Funkcja wywoływana po wywołaniu akcji
        enabled_flag=True,      # Akcja jest aktywna (klikalna), domyślnie tak
        add_to_menu=True,       # Ikona w menu, domyślnie tak
        add_to_toolbar=True,    # Ikona na pasku narzędzi, domyślnie tak
        status_tip=None,        # Tekst podpowiedzi w dymku, domyślnie brak
        whats_this=None,        # Tekst w pasku statusu, domyślnie brak
        parent=None):           # Kontrolka rodzic dla akcji, domyślnie brak
        ...
        action = QAction(icon, text, parent)
        ...
        return action
```

Metoda ta zwraca instancję klasy **`QAction`**. Można ją dalej wykorzystać w celu dalszej konfiguracji przycisku. Można m.in. dodać możliwość pozostawienia przycisk w formie wciśniętej (aktywnej) ustawiając jego właściwość **`setCheckable`** na prawdę logiczną. W ten sposób w QGIS oznaczane są np. aktywne narzędzia mapy (np. Przesuń widok, Informacje o obiekcie), dzięki czemu użytkownik wie jakim narzędziem pracuje.

```
# Stworzenie nowej akcji
akcja = self.add_action( ... )
# Ustawienie możliwości pozostawienia wciśniętego przycisku
akcja.setCheckable( True )
```

Ćwiczenie

Treść zadania

Dodaj nową akcję wykorzystując metodę **`add_action`**. Do akcji powinna zostać dodana nowa ikona *punkt.png* oraz opis *Wybierz punkt na mapie*. Jako funkcję wywoływaną przez tę akcję utwórz w głównej klasie wtyczki metodę **`aktywujNarzedzie`**. Metoda ta powinna drukować tekst *Kliknięto akcję*. Akcja powinna mieć możliwość pozostawania w formie wciśniętej oraz nie jest widoczna w menu wtyczki.

```
git checkout v_6
```

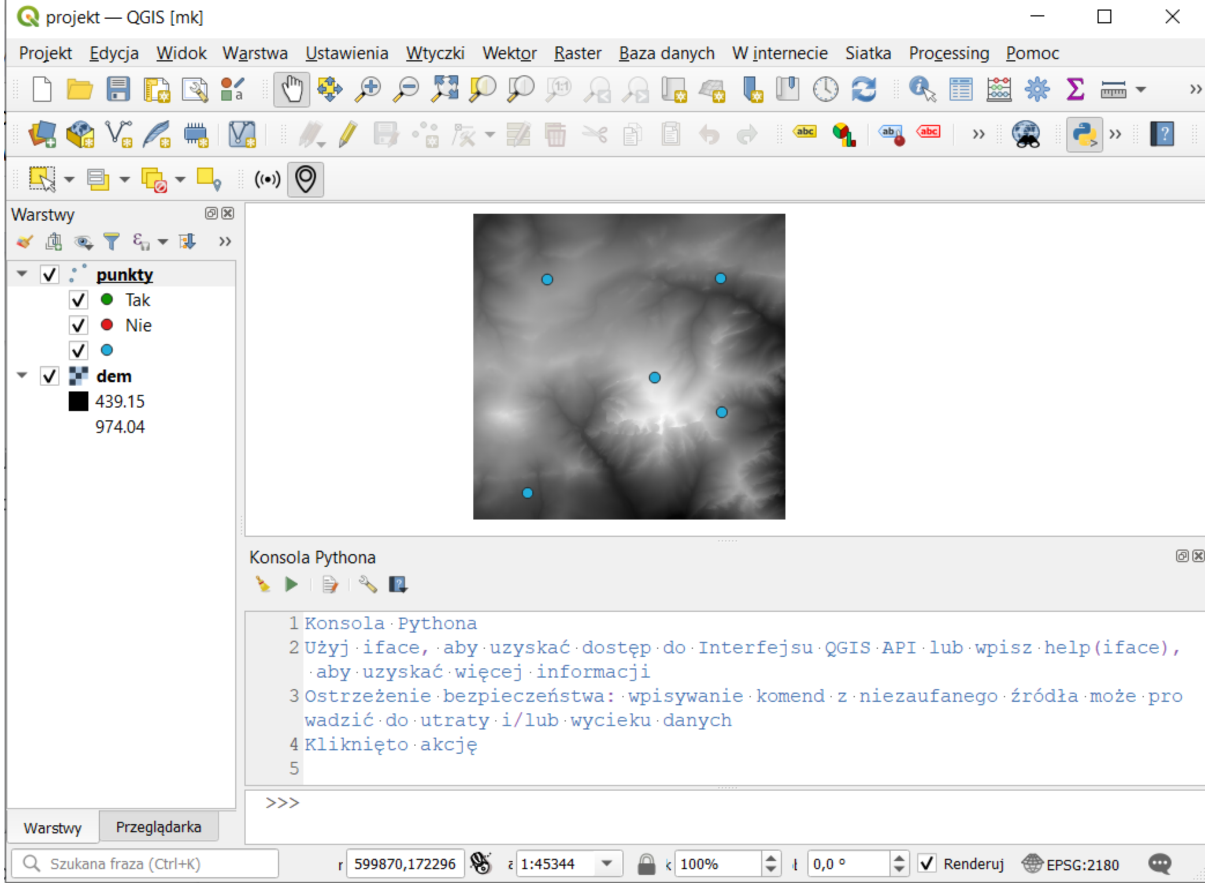
Opis

Na początek należy skopiować do katalogu wtyczki plik *punkt.png*, dopisać ją w zasobach *resources.qrc* oraz skompilować.

Kolejnym krokiem jest zmodyfikowanie głównej klasy wtyczki. Nowe akcje należy definiować w metodzie **`initGui`**. Znajduje się w niej już kod dodający pierwszą akcję odpowiedzialną za wyświetlanie panelu dokowanego. Podobnie należy dodać nowy przycisk, podając dane zgodnie z opisem.

W związku z tym, że przycisk ma pozostawać wciśnięty trzeba go będzie dodatkowo zmodyfikować. W tym celu wynik działania metody **`add_action`** należy przypisać do zmiennej i ustawić jej metodę **`setCheckable`** na prawdę logiczną. Na koniec należy dodać nową metodę w klasie wtyczki, która drukuje określony tekst.

Po zakończeniu kodowania należy odświeżyć wtyczkę w QGIS, powinien pojawić się nowy przycisk na pasku narzędzi. Następnie należy uruchomić *Konsolę Pythona* i po kliknięciu nowego przycisku sprawdzić czy tekst z metody **`aktywujNarzedzie`** pojawi się w niej.



Kod źródłowy

Plik *resources.py*

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

Plik *analiza_zasiegu.py*

```
class AnalizaZasiegu:

    def initGui(self):
        ...
        # Stworzenie nowej akcji
        akcja = self.add_action(
            ':/plugins/analiza_zasiegu/punkt.png',
```

```
            'Wybierz punkt na mapie',
            self.aktywujNarzedzie,
            add_to_menu=False,
            parent=self.iface.mainWindow()
        )
        # Ustawienie możliwości pozostawienia wciśniętego przycisku
        akcja.setCheckable( True )

    def aktywujNarzedzie( self ):
        """ Metoda wywoływana w momencie kliknięcia przycisku akcji """
        print( 'Kliknięto akcję' )
```

Narzędzia mapy

Narzędzia mapy służą do interakcji użytkownika z mapą QGIS. QGIS ma wiele narzędzi wbudowanych np. identyfikacji, zaznaczania czy edycyjne. Możliwe jest tworzenie własnych narzędzi i ich oprogramowanie pod konkretne potrzeby.
W QGIS API znajduje się kilka klas związanych z narzędziami mapy. Wszystkie narzędzia QGIS dziedziczą z głównej klasy **`QgsMapTool`**. Pozostałe klasy dotyczą konkretnych narzędzi i można je podzielić na kilka grup:

- informacja o miejscu kliknięcia - **`QgsMapToolEmitPoint`**,
- zmiana widoku mapy - **`QgsMapToolPan`**, **`QgsMapToolZoom`**,
- identyfikacja/zaznaczanie obiektów - **`QgsMapToolExtent`**, **`QgsMapToolIdentify`**, **`QgsMapToolIdentifyFeature`**,
- edycja danych - **`QgsMapToolCapture`**, **`QgsMapToolAdvancedDigitizing`**, **`QgsMapToolDigitizeFeature`**, **`QgsMapToolEdit`**.

Jednocześnie może być aktywne tylko jedno narzędzie mapy.

QgsMapToolEmitPoint

Jest to jedno z najprostszych narzędzi mapy. Zwraca punkt (instancja klasy **`QgsPointXY`**), w którym użytkownik kliknął na mapę oraz przycisku myszy, który był użyty. Służy do tego sygnał **`canvasClicked`**, do którego można się podpiąć. Współrzędne zwracane są w jednostkach układu projektu QGIS.
Tworząc instancję tej klasy należy podać jako argument instancję klasy **`QgsMapCanvas`** reprezentującą okno mapy, na którym ma być ono używane. W tym celu najprościej wykorzystać obiekt iface.

```
# Import klas z QGIS API
from qgis.gui import QgsMapToolEmitPoint
from qgis.utils import iface


# Stworzenie nowego narzędzia mapy
```

```
narzedzie = QgsMapToolEmitPoint( iface.mapCanvas() )
```

Sygnał **canvasClicked** jest wywoływany w momencie, gdy użytkownik kliknie na mapę. Przekazuje on dwie wartości: współrzędne punktu kliknięcia oraz użyty przycisk myszy. Tak więc, aby podpiąć się pod ten sygnał należy stworzyć metodę, która przyjmuje dwa argumenty.

```
# Funkcja wywoływana w momencie kliknięcia narzędziem na mapie
def klik(punkt, przycisk):
    print( punkt, przycisk )

# Powiązanie sygnału z funkcją
narzedzie.canvasClicked.connect( klik )
```

Aktywacja narzędzi mapy

Aby aktywować narzędzie mapy należy je przekazać do metody **setMapTool** klasy **QgsMapCanvas**. Spowoduje to dezaktywację aktualnego narzędzia mapy.

```
# Aktywacja narzędzia mapy
iface.mapCanvas().setMapTool( narzedzie )
```

Akcje i narzędzia mapy

Narzędzia mapy najwygodniej jest aktywować za pomocą akcji. Taka akcja powinna mieć możliwość pozostawania wciśniętą, dzięki czemu użytkownik jest poinformowany o aktywnym narzędziu. Każde narzędzie mapy (dziedziczące z klasy **QgsMapTool**) ma metodę **setAction**. Jako jej argument należy podać akcję, z którą narzędzie będzie powiązane. Dzięki temu w momencie gdy użytkownik wybierze inne narzędzia akcja wróci do stanu nieaktywnego (nie wciśnięta).
Aktywacja narzędzia powinna odbywać się w metodzie powiązanej z akcją.

```
# Powiązanie narzędzia z wcześniej utworzoną akcją
narzedzie.setAction( akcja )

# Funkcja powiązana z akcją
def akcja(punkt, przycisk):
    # Aktywacja narzędzia mapy
    iface.mapCanvas().setMapTool( narzedzie )
```

Ćwiczenie

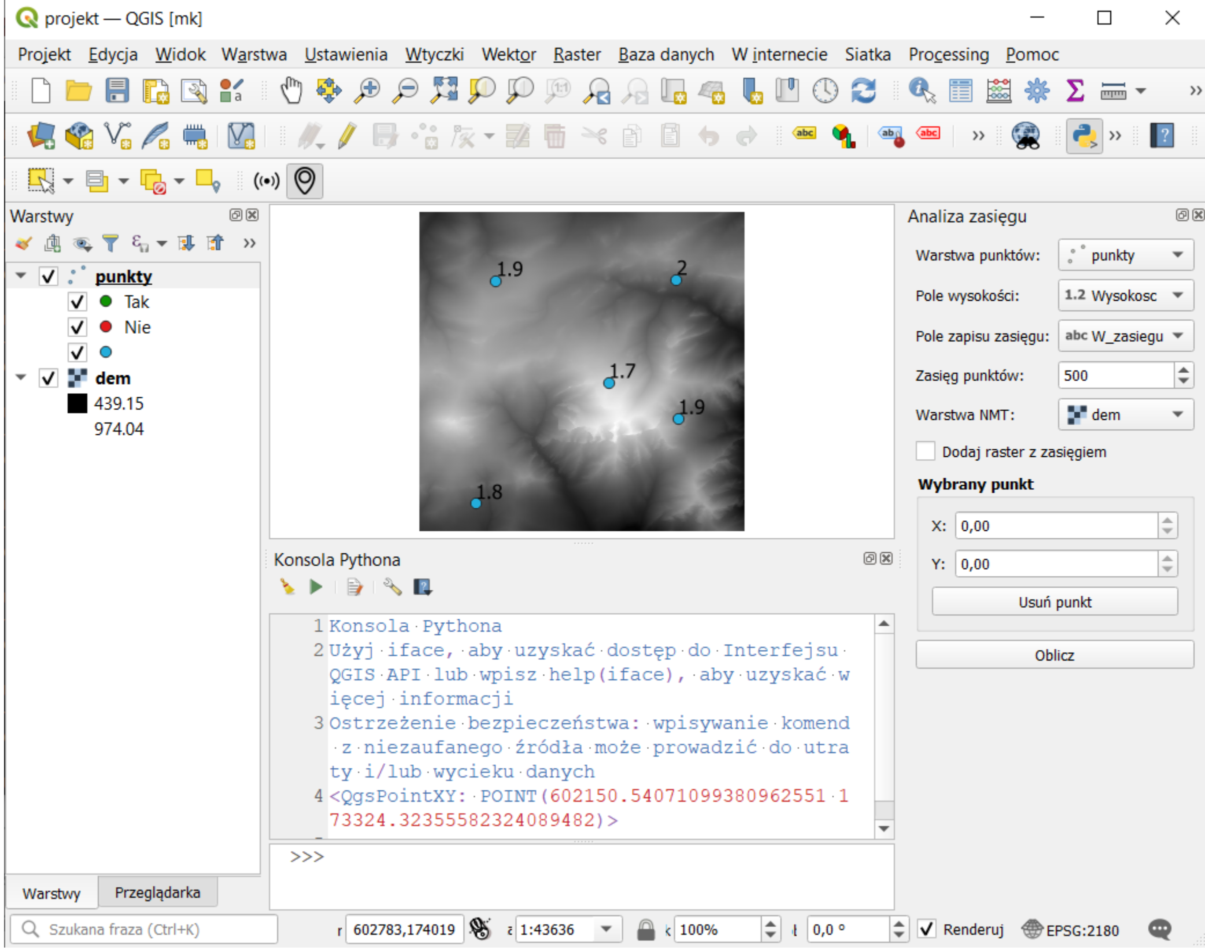
Treść zadania

Dodaj nowe narzędzie mapy do wtyczki wykorzystując klasę **`QgsMapToolEmitPoint`**. Powiąż je z utworzona wcześniej akcją. Sygnał kliknięcia w mapę powiąż z metodą **`wybraniePunktu`**, która będzie znajdować się w klasie reprezentującej panel boczny **`AnalizaZasieguDockWidget`**.

```
git checkout v_7
```

Opis

W pierwszej kolejności należy utworzyć nową instancję klasy **`QgsMapToolEmitPoint`** w metodzie **`initGui`**. Obiekt ten będzie wywoływany w metodzie **`aktywujNarzedzie`**, dlatego należy go zapamiętać jako atrybut klasy wtyczki. Klasa **`QgsMapToolEmitPoint`** znajduje się w module **`qgis.gui`** i należy ją zaimportować. Następnie przypisujemy do narzędzia wcześniej utworzoną akcję za pomocą metody **`setAction`**.
Kolejnym krokiem jest aktywacja narzędzia, odbywa się to w metodzie **`aktywujNarzedzie`**, która jest powiązana z przypisaną akcją. Dodajemy w niej aktywację narzędzia z poziomu klasy **`QgsMapCanvas`** za pomocą metody **`setMapTool`**. To pozwala włączać i wyłączać narzędzie. Działanie należy zweryfikować w QGIS.
Następnie dodajemy obsługę kliknięcia użytkownika na mapie, w momencie gdy narzędzie jest aktywne. W tym celu należy powiązać jego sygnał **`canvasClicked`** z metodą, która ma być dodana w klasie panelu. Takie powiązanie może odbyć się dopiero po tym jak jest utworzona instancja tej klasy. W tym celu należy znaleźć miejsce w kodzie głównej klasy wtyczki gdzie to się odbywa tj. w metodzie **`run`**. Pod linijką z tworzeniem instancji należy dopisać powiązanie sygnału z metodą **`wybraniePunktu`**.
Ostatnim etapem jest dodanie metody **`wybraniePunktu`** do klasy **`AnalizaZasieguDockWidget`**. Można w niej wydrukować przekazywany punkt, w celu zweryfikowania czy narzędzie działa poprawnie.



Kod źródłowy

Plik *analiza_zasiegu.py*:

```
# Import klas z QGIS API
from qgis.gui import QgsMapToolEmitPoint

class AnalizaZasiegu:

    def initGui(self):
        ...
        # Utworzenie nowego narzędzia mapy
        self.narzedzieMapy = QgsMapToolEmitPoint( self.iface.mapCanvas() )
        # Powiązanie narzędzia z akcją, dzięki temu przycisk wróci
        # do normalnej pozycji po wybraniu przez użytkownika
        # innego narzędzia mapy
        self.narzedzieMapy.setAction( akcja )

    def run(self):
```

```
        ...
        if self.dockwidget == None:
            …
            # Wywołanie funkcji po kliknięciu na mapie
            self.narzedzieMapy.canvasClicked.connect(
                self.dockwidget.wybraniePunktu )

    def aktywujNarzedzie( self ):
        """ Metoda wywoływana w momencie kliknięcia przycisku akcji """
        # Aktywacja narzędzia
        self.iface.mapCanvas().setMapTool( self.narzedzieMapy )
```

Plik *analiza_zasiegu_dockwidget.py*:

```
class AnalizaZasieguDockWidget(QtWidgets.QDockWidget, FORM_CLASS):

    def wybraniePunktu(self, punkt, przycisk):
        """ Funkcja wywołana po kliknięciu
        na mapie przez użytkownika """
        print( punkt )
```

Ćwiczenie

Opis

Skonfiguruj kontrolki **`Punkt_X`** i **`Punkt_Y`** aby po wybraniu punktu wyświetlały jego współrzędne. Jeśli nie wybrano żadnego punktu powinny wyświetlać tekst *Brak*. Kliknięcie przycisku *Usuń punkt* czyści oba pola z wcześniejszych ustawień.

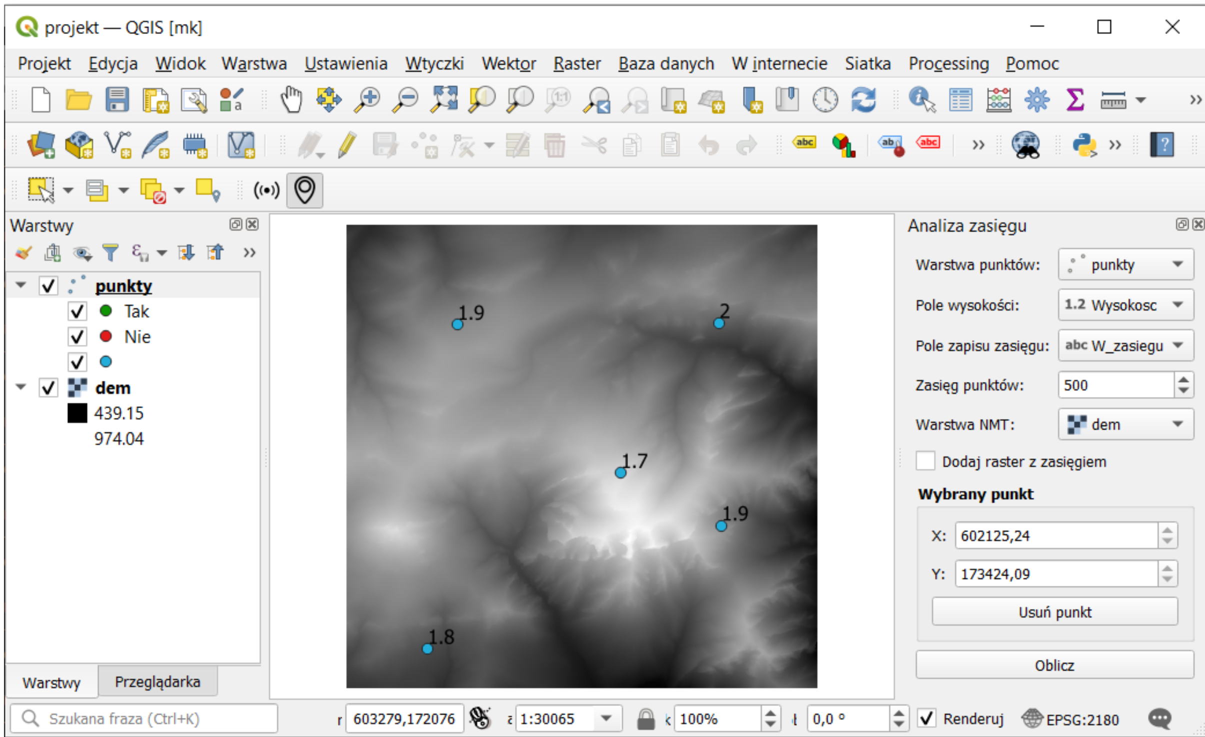
```
git checkout v_8
```

Opis

Przy tworzeniu panelu dokowanego ustawiamy kontrolki **`Punkt_X`** i **`Punkt_Y`** tak, aby przy ustawieniu minimalnej wartości wyświetlany był tekst *Brak*. Wykorzystujemy do tego metodę **`setSpecialValueText`**.

Następnie należy zmodyfikować metodę **`wybraniePunktu`** i ustawić wartości kontrolek wyświetlających współrzędne punktu za pomocą metod **`setValue()`**.

Ostatnim elementem jest oprogramowanie przycisku *Usuń punkt*. W tym celu należy powiązać jego sygnał **`clicked`** z nową metodą **`usunieciePunktu`**. Wewnątrz niej czyścimy obie kontrolki ze współrzędnymi ustawiając ich wartość na *0*.



Kod źródłowy

Plik *analiza_zasiegu_dockwidget.py*

```
from qgis.core import (QgsMapLayerProxyModel, QgsFieldProxyModel, QgsPointXY)

class AnalizaZasieguDockWidget(QtWidgets.QDockWidget, FORM_CLASS):

    def __init__(self, parent=None):
        ...
        # Ustawienie tekstu jeśli nie wybrano punktu
        self.Punkt_X.setSpecialValueText('Brak')
        self.Punkt_Y.setSpecialValueText('Brak')

        # Kliknięcie przycisku Usuń punkt
        self.UsunPunkt.clicked.connect( self.usunieciePunktu )

    def wybraniePunktu(self, punkt, przycisk):
        """ Funkcja wywołana po kliknięciu
        na mapie przez użytkownika """
        # Ustawienie współrzędnych w polach
        self.Punkt_X.setValue( punkt.x() )
        self.Punkt_Y.setValue( punkt.y() )
```

```
    def usunieciePunktu(self):
        """ Usunięcie danych wybranego punktu """
        self.punkt_analizy = None
        # Czyszczenie współrzędnych
        self.Punkt_X.setValue( 0 )
        self.Punkt_Y.setValue( 0 )
```

Rysowanie w oknie mapy

Klasa QgsRubberBand

Do rysowania elementów graficznych na mapie można wykorzystać klasę **QgsRubberBand**. Dzięki niej możliwe jest wstawienie elementów bez konieczności edycji warstw. Z tej klasy korzystają różne narzędzia mapy QGIS np. podczas rysowania nowego obiektu przestrzennego pokazana jest jego geometria, a korzystając z narzędzia identyfikacji wybrany obiekt jest podświetlony. Tworząc instancję klasy **QgsRubberBand** należy przekazać jako argument klasę reprezentującą mapę (**QgsMapCanvas**) oraz typ geometrii (**QgsWkbTypes.PointGeometry**, **QgsWkbTypes.LineGeometry** lub **QgsWkbTypes.PolygonGeometry**).

```
from qgis.gui import QgsRubberBand

z = QgsRubberBand(iface.mapCanvas(), QgsWkbTypes.PointGeometry)
```

Kształt

Aby ustawić lub modyfikować rysowany kształt można wykorzystać kilka metod:

- **setToGeometry** - ustawienie kształtu wg geometrii (**QgsGeometry**), jeśli wcześniej był ustawiony inny kształt zostanie on zastąpiony,
- **addGeometry** - dodanie nowej geometrii (**QgsGeometry**) do istniejącego kształtu, powstanie obiekt wieloczęściowy (*multipart*),
- **addPoint** - dodanie nowego punktu (**QgsPointXY**) do istniejącej geometrii, jeśli aktualny kształt ma więcej niż jedną geometrię można podać jej numer.

Powyższe elementy należy podawać w układzie współrzędnych zgodnym z układem projektu QGIS.

Aby pobrać geometrię z istniejącego znacznika można skorzystać z metody **asGeometry**(), która zwróci geometrię jako instancję klasy **QgsGeometry**. Można również pobrać pojedynczy wierzchołek (**QgsPointXY**) za pomocą metody **getPoint()**. Należy podać numer geometrii oraz jej wierzchołka (domyślnie zwracany jest pierwszy wierzchołek).

Jeśli chcemy usunąć znacznik z mapy można skorzystać z metody **reset()**. Należy jednak pamiętać, że jeśli korzystamy z innego typu geometrii niż linia trzeba podać ten typ jako argument metody. W przeciwnym wypadku po dodaniu nowych wierzchołków kształt będzie rysowany w niepoprawny sposób.

```
z.addPoint( QgsPointXY(604708,173197) )
```

Stylizacja

Klasa **QgsRubberBand** pozwala na prostą stylizację rysowanych elementów (punktów, linii i poligonów). Można to zrobić dedykowanymi metodami, podając jako argumenty żądaną wartość. Są to m.in:

- **setFillColor** - kolor wypełnienia poligonów,
- **setStrokeColor** - kolor punktów, linii i obrysu poligonów,
- **setColor** - kolor znacznika, skrót do jednoczesnego ustawienia wypełnienia poligonów, linii i punktów na ten sam kolor,
- **setIcon** - kształt znacznika dla punktów i wierzchołków linii/poligonów, dostępnych jest kilka rodzajów określonych jako atrybuty klasy **QgsRubberBand**,
- **setIconSize** - rozmiar znacznika dla punktów oraz wierzchołków linii/poligonów podany w pikselach,
- **setWidth** - szerokość linii i obrysów poligonów.

```
z.setIconSize( 20 )
z.setIcon( QgsRubberBand.ICON_X )
z.setWidth( 3 )
```

Kolory

Do ustawiania kolorów należy skorzystać z klas *Qt*. Klasa **QColor** umożliwia zdefiniowanie dowolnego koloru za pomocą różnych systemów zapisu m.in. podając nazwę angielską koloru (*red, green*) lub w formatach *RGB(A), HSV, CMYK*. Przykładowo dla systemu *RGB(A)* wartości określa się podając nasycenie barwami czerwoną (*R* - red), zieloną (*G* - green) i niebieską (*B* - blue) w skali od 0 (brak barwy) do 255 (pełna wartość). Dodatkowo można określić przezroczystość za pomocą czwartej wartości liczbowej (A - kanał alfa) gdzie 0 to pełna przezroczystość, a 255 to jej brak.

```
from qgis.PyQt.QtGui import QColor

# Nazwa koloru
QColor( 'red' )
# Definicja koloru czerwonego w systemi RGB
QColor( 255, 0, 0 )
# Definicja koloru czerwonego z 50% przezroczystością
QColor( 255, 0, 0, 123 )
```

Alternatywnie można skorzystać z modułu Qt, w którym zdefiniowano najczęściej wykorzystywane kolory:

```
from qgis.PyQt.QtCore import Qt
```

```
# Kolor czerwony
Qt.red
```

Ustawienie koloru znacznika:

```
z.setColor( Qt.green )
z.reset()
```

Ćwiczenie

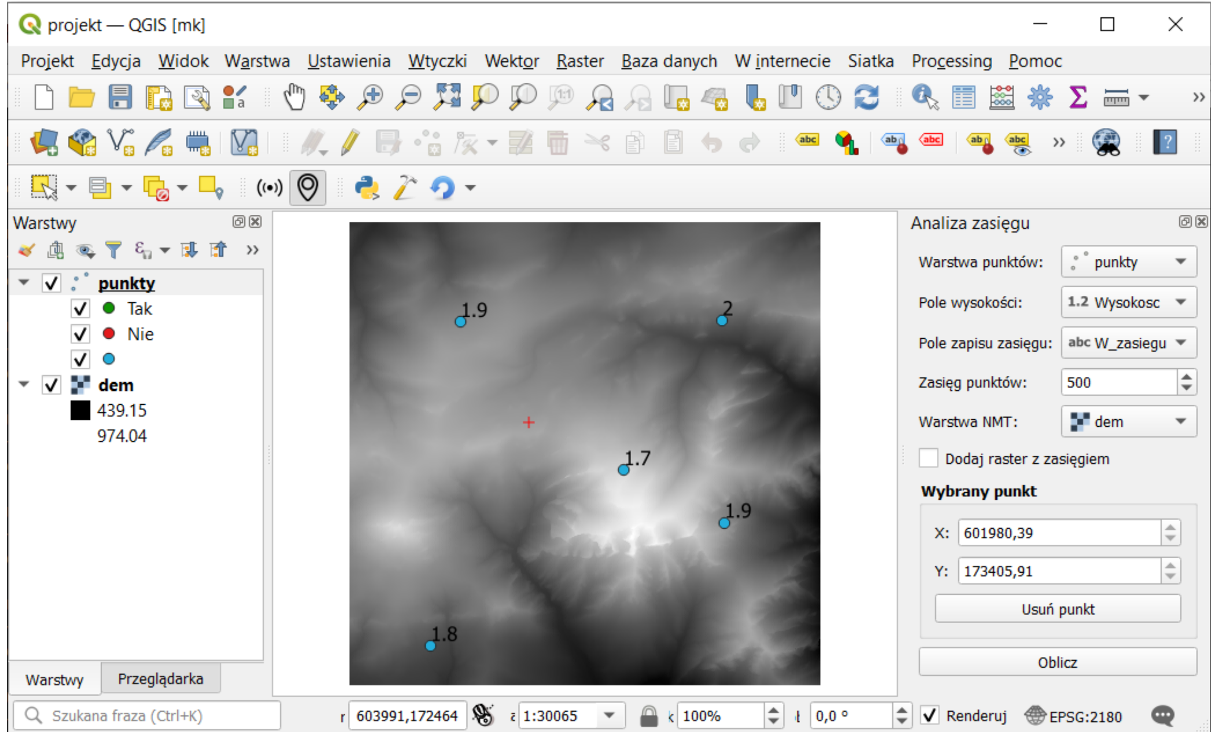
Treść zadania
Wyświetl na mapie w miejscu wskazanym przez użytkownika czerwony krzyżyk o rozmiarze 10. Powinien on zniknąć po kliknięciu przycisku *Usuń punkt* oraz przy wyłączaniu wtyczki.

```
git checkout v_9
```

Opis
Do rysowania wykorzystujemy instancję klasy **`QgsRubberBand`**. Musi ona być zapamiętana w klasie panelu dokowanego ponieważ będziemy z niej korzystać w kilku miejscach. Tworząc instancję znacznika jako argumenty podajemy obiekt mapy oraz typ geometrii - w naszym przypadku jest to **`QgsWkbTypes.PointGeometry`**. Następnie ustawiamy wygląd znacznika:

- **`setColor`** - kolor, dla ułatwienia można skorzystać z predefiniowanych kolorów zdefiniowanych w module **`Qt`** - dla czerwonego jest to **`Qt.red`**.
- **`setIcon`** - kształt, krzyżyk dostępny jest w atrybucie **`QgsRubberBand.ICON_CROSS`**
- **`setIconSize`** - rozmiar.

Nowy znacznik ustawiany jest w momencie kliknięcie przez użytkownika na mapie, w naszym kodzie jest to metoda **`wybraniePunktu`**. Na początku wyczyśćmy wcześniejsze ustawienia punktu wywołując metodę **`usunieciePunktu`**, a następnie dodajmy punkt do naszego znacznika metodą **`addPoint`**. Czynności te należy wykonać na początku wywołania metody **`wybraniePunktu`**, tak aby nie usunąć wcześniej zapamiętanych informacji.
Teraz należy dodać czyszczenie znacznika po kliknięciu przycisku *Usuń punkt*. Na końcu metody **`usunieciePunktu`** dodajemy resetowanie za pomocą metody **`reset`** podając jako atrybut typ geometrii punktowej. Jeśli tego nie zrobimy geometria zostanie ustawiona na liniową.
Ostatnim etapem jest obsłużenie czyszczenia znacznika w momencie gdy wtyczka jest wyłączana. Jeśli tego nie zrobimy to po jej wyłączeniu na mapie pozostanie czerwony krzyżyk i do jego usunięcia konieczne będzie zresetowanie QGIS. Najlepszym na to miejscem jest metoda **`unload`** w głównej klasie wtyczki. Należy tylko pamiętać aby sprawdzić czy panel dokowany faktycznie istnieje. Jeśli tak to możemy wywołać metodę **`usunieciePunktu`**, która wyczyści znacznik.



Kod źródłowy

Plik *analiza_zasiegu_dockwidget.py*

```
from qgis.PyQt.QtCore import pyqtSignal, Qt
# Import klas QGIS API
from qgis.core import QgsMapLayerProxyModel, QgsFieldProxyModel, QgsWkbTypes
from qgis.gui import QgsRubberBand
from qgis.utils import iface

class AnalizaZasieguDockWidget(QtWidgets.QDockWidget, FORM_CLASS):

    def __init__(self, parent=None):
        ...
        # Znacznik wskazanego punktu
        self.znacznik = QgsRubberBand( iface.mapCanvas(),
                                       QgsWkbTypes.PointGeometry )
        self.znacznik.setColor( Qt.red )
        self.znacznik.setIcon( QgsRubberBand.ICON_CROSS )
        self.znacznik.setIconSize( 10 )

    def wybraniePunktu(self, punkt, przycisk):
        """ Funkcja wywołana po kliknięciu
        na mapie przez użytkownika """
```

```
        # Usunięcie informacje o wcześniej wybranym punkcie
        self.usunieciePunktu()
        # Dodanie znacznika mapy
        self.znacznik.addPoint( punkt )
        …

    def usunieciePunktu(self):
        """ Usunięcie danych wybranego punktu """
        …
        # Wyczyszczenie znacznika mapy
        self.znacznik.reset(QgsWkbTypes.PointGeometry)
```

Plik *analiza_zasiegu.py*

```
class AnalizaZasiegu:

    def unload(self):
        ...
        if self.dockwidget != None:
            self.dockwidget.usunieciePunktu()
```

Analizy przestrzenne i automatyzacja procesów przetwarzania danych

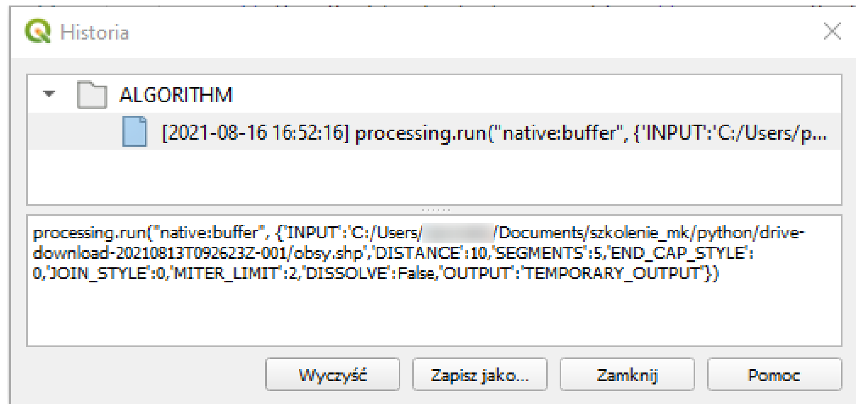
Głównym narzędziem analitycznym QGIS jest wtyczka *Processing*. Integruje ona różne silniki analityczne m.in. algorytmy natywne, GDAL, GRASS GIS, SAGA GIS. Narzędzia uruchamiane są w ujednoliconym interfejsie graficznym, dzięki czemu użytkownik nie musi znać specyfiki pracy w danej aplikacji. Możliwe jest dodawanie nowych algorytmów poprzez instalację wtyczek integrujących inne aplikacje operujące na danych przestrzennych lub poprzez tworzenie skryptów w języku Python. Pojedyncze algorytmy można ze sobą łączyć w kreatorze modeli tworząc bardziej złożone systemy.
W dalszej pracy będziemy korzystać z wtyczki *Processing*, która jest dostarczona razem z instalatorem QGIS. Jeśli nie jest widoczne menu *Processing* należy włączyć tą wtyczkę.
Wszystkie dostępne w panelu *Processing* algorytmy można wywołać z poziomu języka Python w dowolnym miejscu QGIS. Służy do tego moduł **`processing`**:

```
from qgis import processing
```

Dzięki temu mechanizmowi możliwe jest zautomatyzowanie procesów związanych z przetwarzaniem różnego rodzaju danych. Częściowo można to osiągnąć korzystając z modeli lub przetwarzania wsadowego czyli funkcjonalności dostępnych we wtyczce *Processing*. Ale mają one swoje ograniczenia lub korzystanie z niektórych rozwiązań w codziennej pracy może być zbyt pracochłonne lub skomplikowane. Przykładem może być wielokrotne wykonywanie tej samej operacji na pojedynczych obiektach warstwy wektorowej czy wykonanie pre- lub post-processingu analizowanych danych. W takim wypadku warto skorzystać z możliwości jakie daje język Python poprzez wywoływanie z jego poziomu dostępnych algorytmów. W ten sposób można stworzyć narzędzia, które są proste w użyciu dla użytkowników i dają dużą elastyczność np. poprzez określenie dodatkowych elementów, które nie są dostępne w używanych algorytmach (np. filtrowanie danych). Zawsze jednak warto rozważyć czy daną automatyzację procesu można osiągnąć za pomocą dedykowanych narzędzi Processingu lub QGIS czy jednak warto zainwestować czas w stworzenie dedykowanego narzędzia np. w postaci wtyczki.

Informacje o algorytmach

Aby uruchomić algorytm z poziomu kodu Pythona należy znać jego identyfikator oraz nazwy poszczególnych parametrów. Najprostszym sposobem na uzyskanie tych informacji jest normalne wykonanie danego algorytmu i sprawdzenie jego wywołania w historii. W tym celu z menu *Processing* wybieramy pozycję *Historia*. Okno podzielone jest na dwie części. W górnej jest lista ostatnio wykonanych algorytmów. Po zaznaczeniu dowolnego z nich w dolnym polu pojawi się informacja o jego wywołaniu. Jest to gotowy kod, który można wkleić do Konsoli Pythona i po zatwierdzeniu całość zostanie wykonana tak samo jak przy zwykłym uruchomieniu.



Funkcja **processing.run** przyjmuje dwa argumenty. Pierwszym jest identyfikator algorytmu. Każdy algorytm w *Processingu* ma unikalną nazwę. Drugi argument to słownik, którego elementami są pary z nazwą parametru i jego wartością.
Aby uzyskać więcej informacji o danym algorytmie oraz jego parametrach można skorzystać z funkcji **processing.algorithmHelp** podając jako argument identyfikator algorytmu:

```
processing.algorithmHelp( "native:buffer" )
```

Pojawi się opis danego algorytmu. Szczególnie warto zwrócić uwagę na część *Input parameters* gdzie opisane są wszystkie parametry.

```
INPUT: Warstwa wejściowa

   Parameter type:   QgsProcessingParameterFeatureSource

   Accepted data types:
         - str: ID warstwy
         - str: nazwa warstwy
         - str: źródło warstwy
         - QgsProcessingFeatureSourceDefinition
         - QgsProperty
         - QgsVectorLayer
```

Powyższy tekst opisuje warstwę wejściową algorytmu *Bufor*. Ma on nazwę *INPUT* i jako wartość może przyjąć kilka rodzajów danych m.in. ścieżkę do pliku lub instancję klasy **QgsVectorLayer**.
Jeśli wynikiem analizy jest warstwa to jako parametr wyjściowy można podać ścieżkę do pliku, który zostanie utworzony lub wpisać **TEMPORARY_OUTPUT**. W tym drugim przypadku QGIS utworzy warstwę tymczasową. Jest to przydatne szczególnie gdy utworzona warstwa będzie dalej przetwarzana i nie jest wymagane jej zapisanie do konkretnej lokalizacji.

Wywoływanie algorytmów z poziomu języka Python

Aby uruchomić algorytm należy podać niezbędne informacje w metodzie processing.run. Podczas wykonywania analizy interpreter Pythona będzie czekał na jej zakończenie. Jako wynik zwrócony zostanie słownik zawierający informacje o parametrach wyjściowych np. utworzonej warstwie.

```
# Wywołanie algorytmy Bufor
wynik = processing.run("native:buffer",
    {'INPUT':'plik.shp',
    'DISTANCE':10,
    'SEGMENTS':5,
    'END_CAP_STYLE':0,
    'JOIN_STYLE':0,
    'MITER_LIMIT':2,
    'DISSOLVE':False,
    'OUTPUT':'TEMPORARY_OUTPUT'
})
```

Ćwiczenie

Treść zadania

Stwórz metodę **`analizaZasiegu`** w klasie panelu dokowanego, która będzie wywoływana po wciśnięciu przycisku *Oblicz*. Jeśli użytkownik wskazał punkt na mapie należy wydrukować wszystkie obiekty warstwy punktowej.

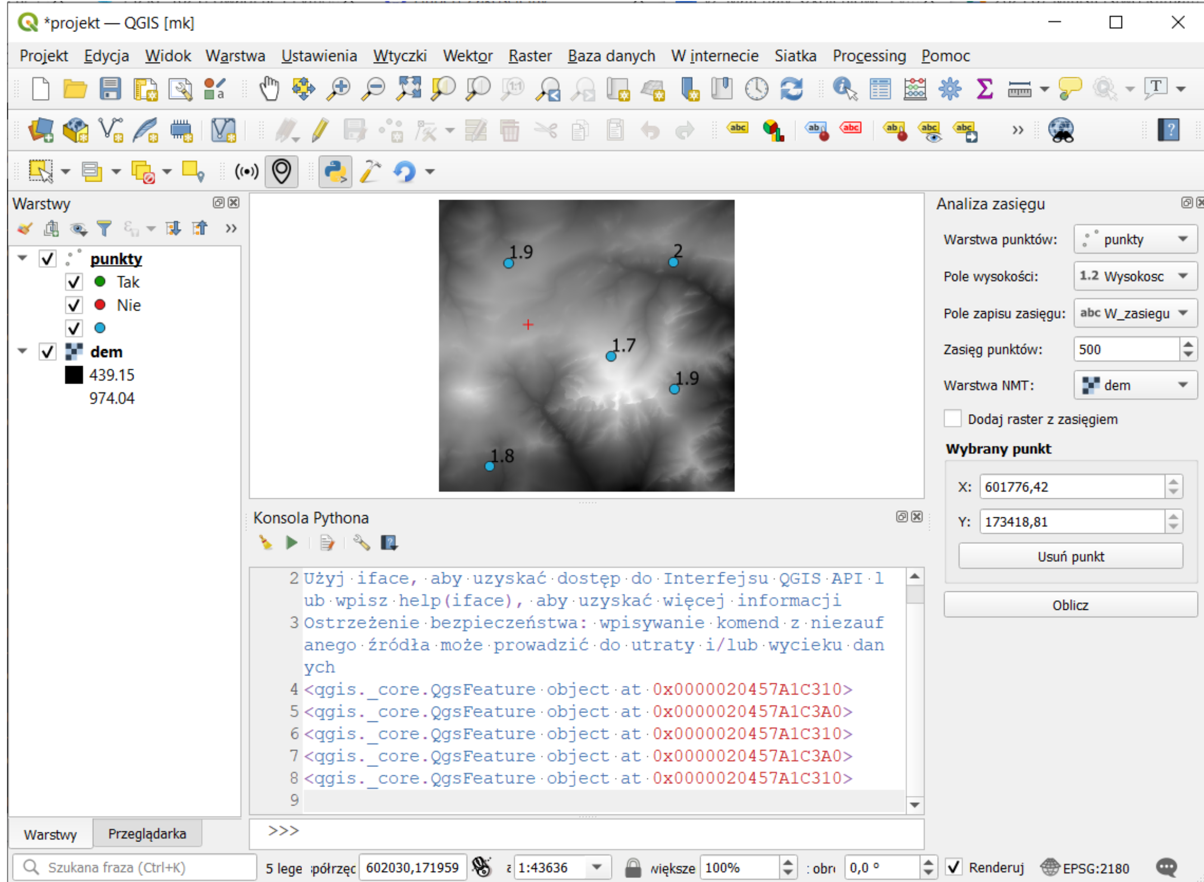
```
git checkout v_10
```

Opis

Należy stworzyć nową metodę **`analizaZasiegu`** w klasie **`AnalizaZasieguDockWidget`** i połączyć ją z sygnałem **`clicked`** przycisku *Oblicz*.

Analiza ma być wykonana jeśli użytkownik wybrał punkt. W związku z tym na początku należy sprawdzić czy punkt jest faktycznie ustawiony. Łatwo można to uzyskać pobierając pierwszy punkt znacznika za pomocą metody **`getPoint`**. Jeśli punkt nie został ustawiony to metoda ta zwróci **`None`**, w przeciwnym wypadku otrzymamy instancję klasy **`QgsPointXY`**.

Następnie należy pobrać informację o wybranej warstwie punktowej. Służy do tego metoda **`currentLayer()`** klasy **`QgsMapLayerComboBox`**, która zwraca instancję klasy **`QgsVectorLayer`**. Następnie za pomocą metody **`getFeatures()`** w pętli **`for`** iterujemy po obiektach przestrzennych warstwy i drukujemy je w konsoli.



Kod źródłowy

Plik *analiza_zasiegu_dockwidget.py*

```
class AnalizaZasieguDockWidget(QtWidgets.QDockWidget, FORM_CLASS):
    def __init__(self, parent=None):
        ...
        # Rozpoczęcie analizy
        self.Oblicz.clicked.connect( self.analizaZasiegu )

    def analizaZasiegu(self):
        """ Wykonanie analizy zasięgu """
        punkt = self.znacznik.getPoint(0)
        if punkt is None:
            # Brak wybranego punktu analizy
            return
        wektor = self.Punkty.currentLayer()
        # Iteracja po obiektach warstwy punktowej
        for obiekt in wektor.getFeatures():
            print( obiekt )
```

Ćwiczenie

Treść zadania

Dla każdego obiektu punktowego wykonaj algorytm *Pole widzenia* z grupy *GDAL - Raster różne* wtyczki *Processing*. Wczytaj wszystkie wynikowe warstwy rastrowe do QGIS.

```
git checkout v_11
```

Opis

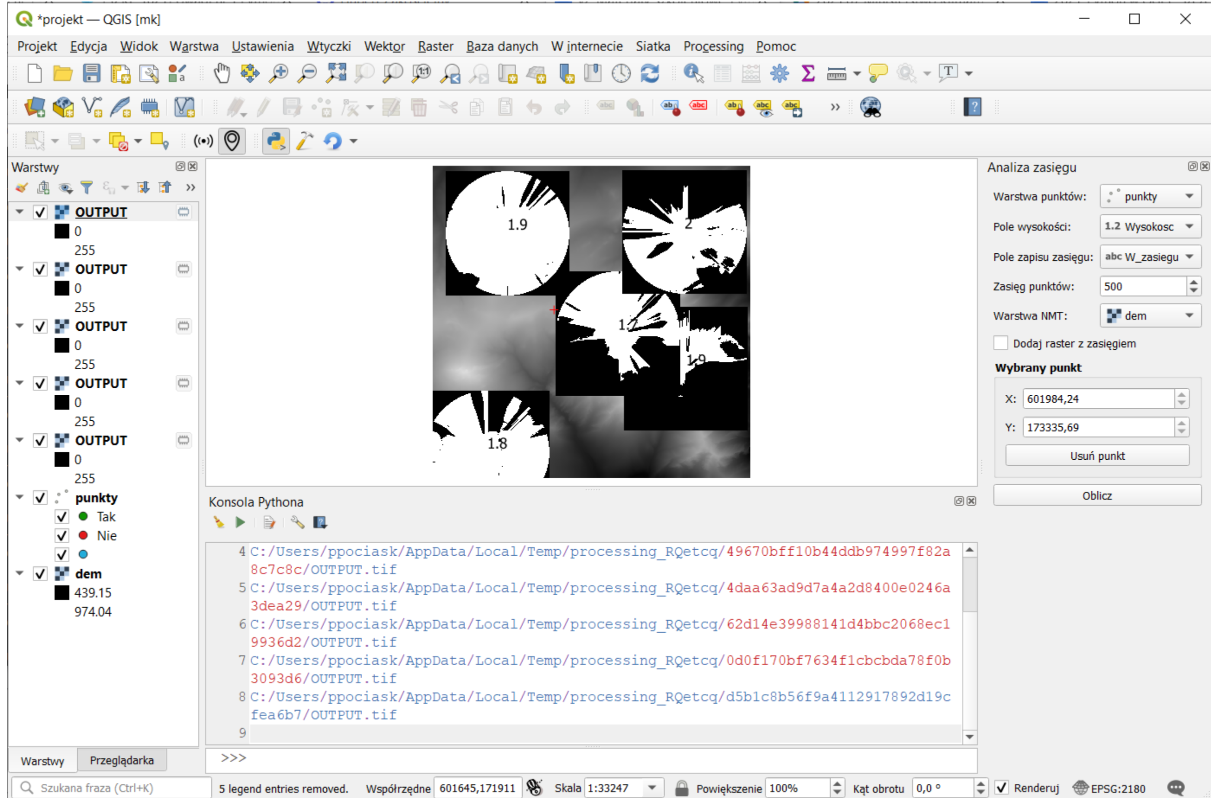
W pierwszej kolejności należy uruchomić algorytm *Pole widzenia* dla przykładowych ustawień. Następnie, po wejściu w menu *Processing* -> *Historia* odszukać na liście wywołanie algorytmu i skopiować polecenie z dolnej części okna.

Polecenie to należy wkleić do nowej metody **`analizujPunkt`**, która jako argument przyjmie przetwarzany aktualnie obiekt przestrzenny warstwy punktowej. Dostosowujemy parametry algorytmu:

- **`INPUT`** - warstwa rastrowa NMT
- **`OBSERVER`** - geometria obiektu przestrzennego
- **`OBSERVER_HEIGHT`** - wysokość pobrana z pola **`Wysokosc`** obiektu
- **`MAX_DISTANCE`** - maksymalny zasięg z widżetu **`Zasieg`**.

Jako wynik funkcji zwracamy ścieżkę do utworzonego nowego rastra (klucz **`OUTPUT`** wynikowego słownika).

W pętli iterującej po obiektach warstwy punktowej należy wywołać stworzoną metodę podając jako argument aktualny obiekt i zapisując jej wynik do zmiennej. Przechowuje ona ścieżkę do utworzonego pliku. Warstwę można wczytać do QGIS za pomocą metody **`iface.addRasterLayer`**.



Kod źródłowy

Plik *analiza_zasiegu_dockwidget.py*

```
# Import klas QGIS API
import processing

class AnalizaZasieguDockWidget(QtWidgets.QDockWidget, FORM_CLASS):
    def analizaZasiegu(self):
        # Iteracja po obiektach warstwy punktowej
        for obiekt in wektor.getFeatures():
            plik_zasiegu = self.analizujPunkt( obiekt )
            print( plik_zasiegu )
            iface.addRasterLayer( plik_zasiegu )

    def analizujPunkt(self, obiekt):
        result = processing.run('gdal:viewshed', {
            'INPUT': self.NMT.currentLayer(),
            'BAND': 1,
            'OBSERVER': obiekt.geometry(),
            'OBSERVER_HEIGHT': obiekt[self.Wysokosc.currentField()],
            'TARGET_HEIGHT': 1,
            'MAX_DISTANCE': self.Zasieg.value(),
            'OUTPUT': 'TEMPORARY_OUTPUT'
        })
        return result['OUTPUT']
```

Ćwiczenie

Treść zadania

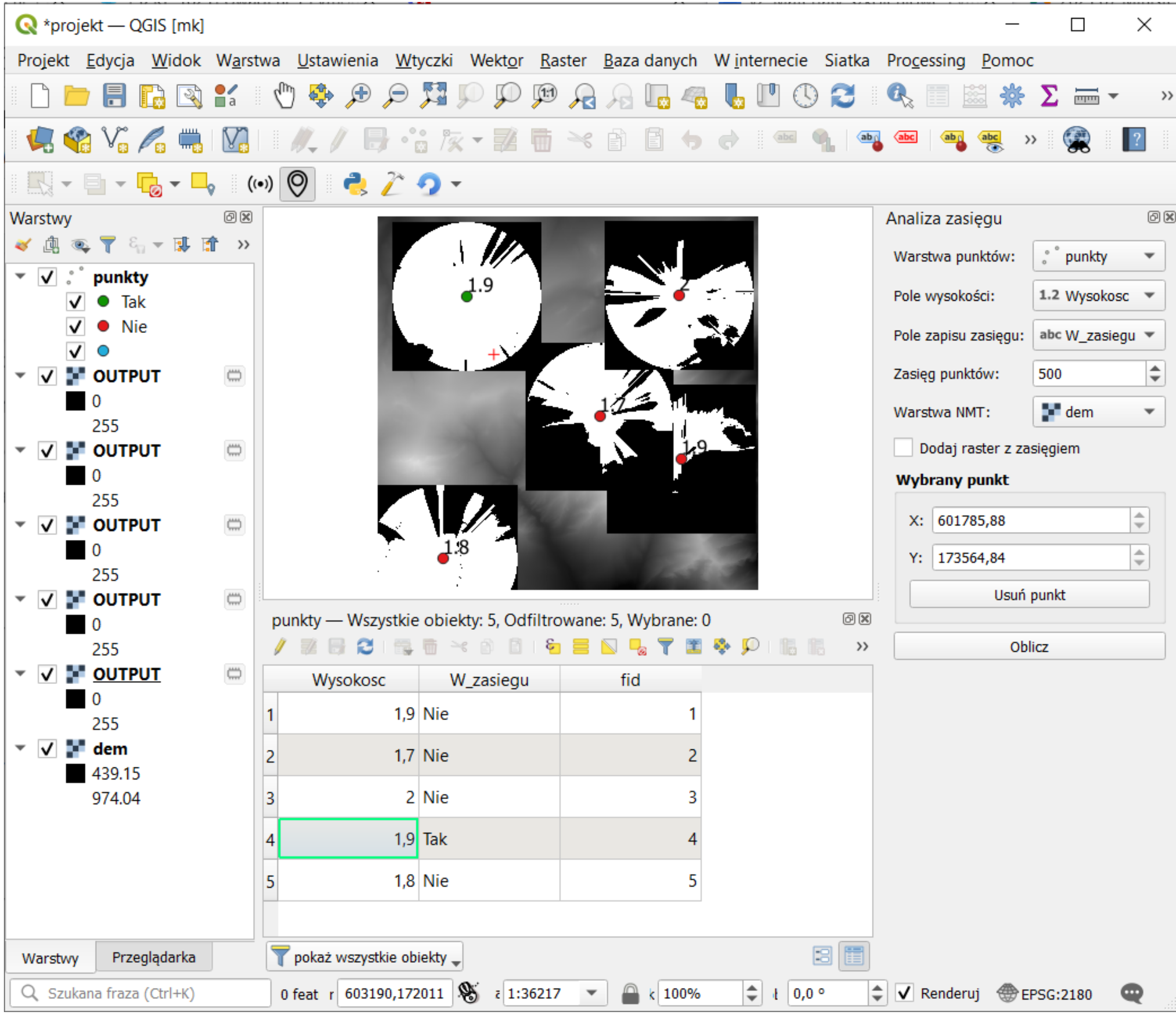
Dla każdego wyliczonego obszaru sprawdź czy wskazany przez użytkownika punkt analizy znajduje się w jego zasięgu. Odpowiednią informację zapisz w atrybucie **`PoleZasiegu`** obiektu punktowego.

```
git checkout v_12
```

Opis

Aby sprawdzić czy punkt analizy znajduje się w zasięgu danego obiektu należy sprawdzić wartość wyliczonego rastra w tym miejscu. Analiza *Pole widzenia* zwraca wartość 255 jeśli dany piksel jest w zasięgu obserwatora, w przeciwnym wypadku zwraca 0. Do odczytu wartości rastra należy skorzystać z metody identify klasy **`QgsRasterDataProvider`**.

Zwraca ona słownik, którego kluczami są numery kanałów rastra (zaczynając od 1), a wartościami odczytane dane z piksela w tym punkcie.
Mając odczytane informacje należy zmodyfikować wartość pola w warstwie punktowej. Służy do tego metoda **`changeAttributeValues`**, która przyjmuje słownik. Jego kluczem jest identyfikator obiektu (**`QgsFeature.id()`**), a wartością kolejny słownik, gdzie kluczem jest indeks pola, a wartością zmienna do zapisania. W kontrolce **`PoleZasiegu`** możemy zwrócić nazwę wybranego przez użytkownika atrybutu. Aby uzyskać jego indeks w tabeli atrybutów warstwy punktowej musimy skorzystać z metody **`indexFromName`**.
Na koniec należy odświeżyć informacje w warstwie punktowej aby QGIS ponownie ją narysował z uwzględnieniem nowych wartości atrybutów za pomocą metody **`reload`**.



Kod źródłowy

Plik *analiza_zasiegu_dockwidget.py*

```
# Import klas QGIS API
from qgis.core import (QgsMapLayerProxyModel, QgsFieldProxyModel, QgsWkbTypes,
    QgsRasterLayer, QgsRaster)

class AnalizaZasieguDockWidget(QtWidgets.QDockWidget, FORM_CLASS):
```

```
    def analizaZasiegu(self):
        …
        # Pobranie informacji o polu do zapisu informacji
        pole_zapisu = self.PoleZasiegu.currentField()
        indeks_pola_zapisu = wektor.fields().indexFromName( pole_zapisu )

        # Iteracja po obiektach warstwy punktowej
        for obiekt in wektor.getFeatures():
            …
            # Obiekt reprezentujący utworzony raster
            raster_zasiegu = QgsRasterLayer( plik_zasiegu )
            # Identyfikacja wartości rastra w punkcie
            wynik = raster_zasiegu.dataProvider().identify(
                        punkt,
                        QgsRaster.IdentifyFormatValue ).results()
            # Czy kanał 1 rastra zawiera informacje
            if wynik[1]:
                w_zasiegu = 'Tak'
            else:
                w_zasiegu = 'Nie'
            wektor.dataProvider().changeAttributeValues(
                        { obiekt.id(): {indeks_pola_zapisu: w_zasiegu}})

        # Odświeżenie warstwy punktowej
        wektor.reload()
```

Ćwiczenie

Treść zadania

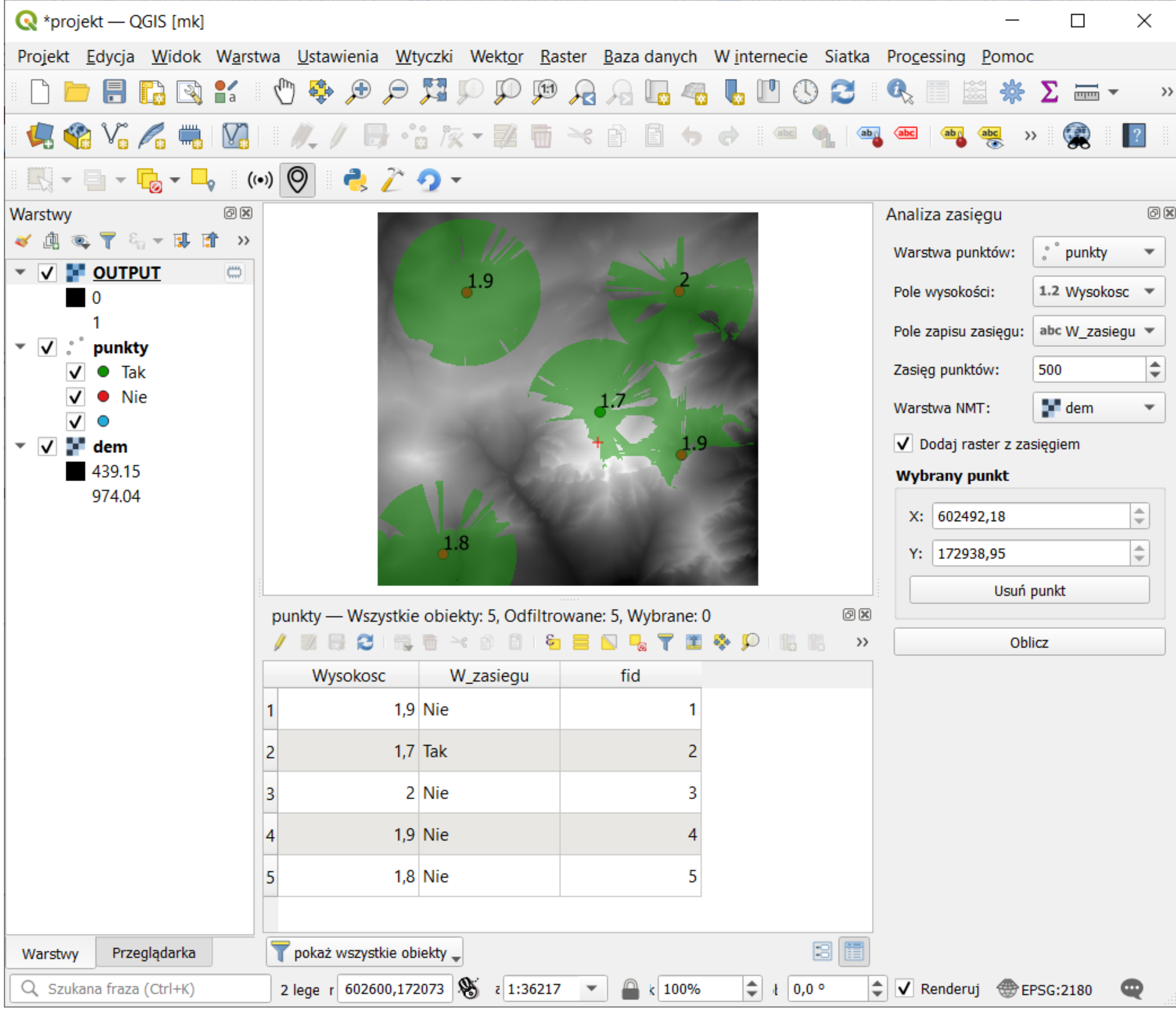
Jeśli użytkownik zaznaczy opcję *Dodaj raster z zasięgiem* za pomocą algorytmu *Raster boolean OR* (w grupie *Raster - analiza*) połącz cząstkowe rastry z zasięgiem w jeden i wczytaj go do QGIS. Styl wczytanej warstwy należy wczytać z pliku *zasieg.qml*, który znajduje się w katalogu ćwiczeniowym.

```
git checkout v_13
```

Opis

Aby wywołać algorytm *Raster boolean OR* z poziomu kodu w pierwszej kolejności należy uruchomić go ręcznie. Jako warstwy wejściowe ustawiamy warstwy zasięgów utworzone i wczytane przy wcześniejszym wywołaniu wtyczki. Jako warstwę odniesienia należy wskazać numeryczny model terenu, tak, żeby wynikowy plik miał rozmiar całego analizowanego obszaru. Dodatkowo należy zaznaczyć opcję *Traktuj brak danych jako fałsz*, dzięki czemu w

wynikowym rastrze obszary poza cząstkowymi zasięgami będą traktowane jako puste. Po zakończeniu działania można podejrzeć w historii *Processingu* wywołanie tego algorytmu.
Jako parametr INPUT należy podać listę łączonych zasięgów cząstkowych, należy więc zapisać ścieżki do plików do obiektu listowego. Po zakończeniu iteracji po obiektach warstwy punktowej sprawdzamy czy użytkownik zaznaczył opcję *Dodaj raster z zasięgiem* za pomocą metody **`isChecked`**. Jeśli tak to uruchamiamy analizę przekazując utworzoną wcześniej listę. Jako parametr *REF_LAYER* podajemy raster z numerycznym modelem terenu. Wynik działania algorytmu wczytujemy do QGIS za pomocą metody **`addRasterLayer`**.
Kolejnym krokiem jest automatyczne przypisanie stylu z pliku *zasieg.qml*. W tym celu należy skopiować ten plik do katalogu wtyczki. Następnie należy określić ścieżkę do tego pliku z poziomu kodu. W tym celu można skorzystać ze specjalnej zmiennej **`__file__`**, która przechowuje ścieżkę do aktualnego pliku *.py*. Wykorzystując moduł os.path można zdefiniować ścieżkę do pliku *zasieg.qml*, który jest w tym samym katalogu co plik *.py*. Do wczytania pliku QML jako stylu warstwy służy metoda **`loadNamedStyle`**.



Kod źródłowy

Plik *analiza_zasiegu_dockwidget.py*

```python
class AnalizaZasieguDockWidget(QtWidgets.QDockWidget, FORM_CLASS):
    def __init__(self, parent=None):
        # Plik stylu rastar z całkowitym zasięgiem
        self.plik_stylu = os.path.join(
                    os.path.dirname(__file__), 'zasieg.qml' )

    def analizaZasiegu(self):
        …
        # Lista ze stworzonymi rastrami
        lista_zasiegow = []

        # Iteracja po obiektach warstwy punktowej
        for obiekt in wektor.getFeatures():
            plik_zasiegu = self.analizujPunkt( obiekt )
            lista_zasiegow.append( plik_zasiegu )
            …
        # Dodanie warstwy z całkowitym zasięgiem
        if self.DodajRaster.isChecked():
            self.oblicz_zasieg( lista_zasiegow )

    def oblicz_zasieg(self, lista_zasiegow):
        """ Obliczenie całkowitego zasięgu """
        result = processing.run('native:rasterlogicalor', {
            'INPUT': lista_zasiegow,
            'REF_LAYER': self.NMT.currentLayer(),
            'NODATA_AS_FALSE': True,
            'OUTPUT':'TEMPORARY_OUTPUT'
        })
        # Dodanie rastra do mapy
        raster = iface.addRasterLayer( result['OUTPUT'] )
        # Ustawienie stylu z pliku QML
        raster.loadNamedStyle( self.plik_stylu )
```