Sfinansowano ze środków Narodowego Funduszu Ochrony Środowiska i Gospodarki Wodnej

Ministerstwo Klimatu i Środowiska

OnGeo

Python w QGIS

Poziom średniozaawansowany

MATERIAŁY SZKOLENIOWE

Spis treści

1. Informacje wstępne

Do przygotowania skryptu wykorzystane zostały dane pochodzące z następujących zbiorów:

- Numeryczny Model Terenu,
- Dane Państwowego Rejestru Granic,
- Dane GDOŚ,
- Dane Państwowego Zasobu Geodezyjnego i Kartograficznego,
- Materiały szkoleniowe dostępne na stronie ekoportal.gov.pl.

Do przygotowania ćwiczeń wykorzystany został program QGIS Białowieża w wersji LTR o numerze 3.22.10.

Wykorzystywane środowisko Python: Python 3.9.5.

2. QGIS – funkcjonalność aplikacji i jej powiązania ze środowiskiem Python

QGIS i PYTHON

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

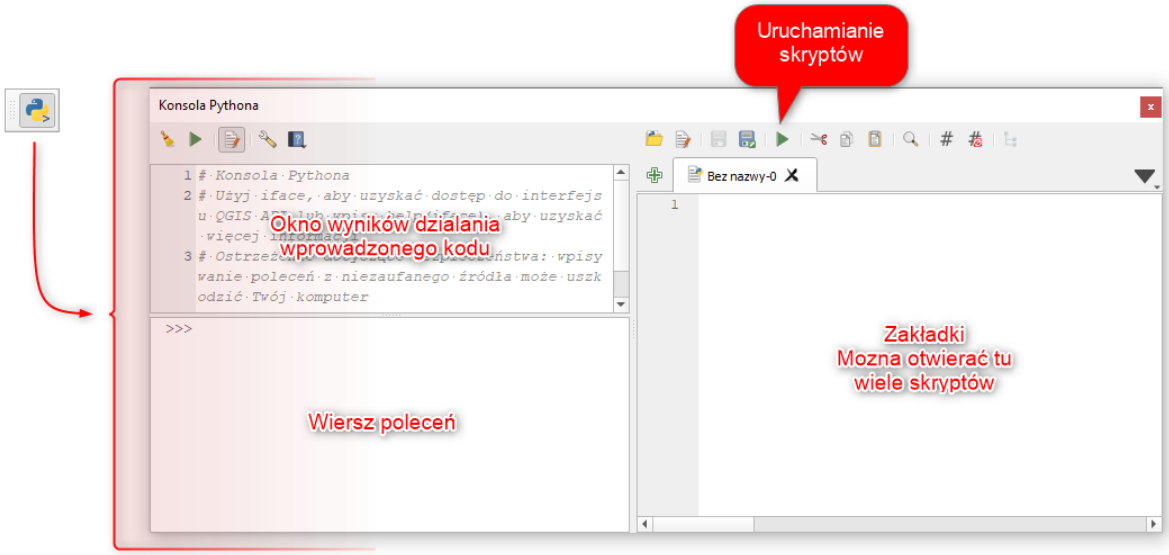
- **Qt** - pakiet bibliotek i narzędzi, służących głównie tworzeniu wieloplatformowych graficznych interfejsów aplikacji (GUI), wykorzystywany m.in. przy tworzeniu wtyczek,
- **GDAL/OGR** - odczyt i zapis rastrowych (GDAL) i wektorowych (OGR) danych przestrzennych zapisanych w różnych formatach,
- **GEOS**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

Oprogramowanie QGIS i środowisko języka Python są silnie zintegrowane. Język ten wykorzystywany jest w aplikacji jako język skryptowy, obsługiwany w jej interfejsie głównie z tzw. Konsoli Python. Narzędzie to służy jako interfejs umożliwiający wprowadzanie kodu Python oraz pracę z bibliotekami obsługi danych przestrzennych (ale nie tylko). 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.

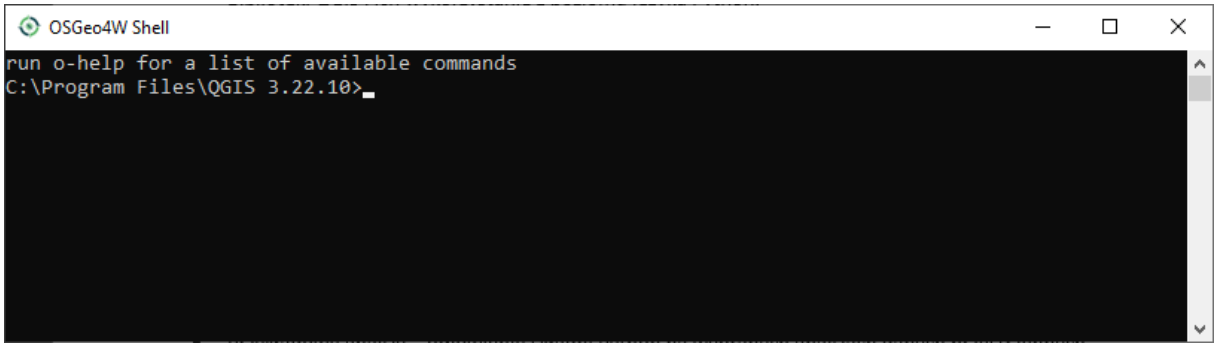
Oba narzędzia są ze sobą zintegrowane i uruchomione we wspólnym środowisku. Oznacza to, że zmienne, importy itp. zdefiniowane w jednym z tych miejsc są również widoczne w drugim. W konsoli automatycznie dostępne są klasy QGIS API i nie ma potrzeby ich importowania. Aby uruchomić konsolę należy w QGIS z menu Wtyczki wybrać polecenie Konsola Pythona lub kliknąć odpowiedni przycisk na pasku narzędzi  .



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.

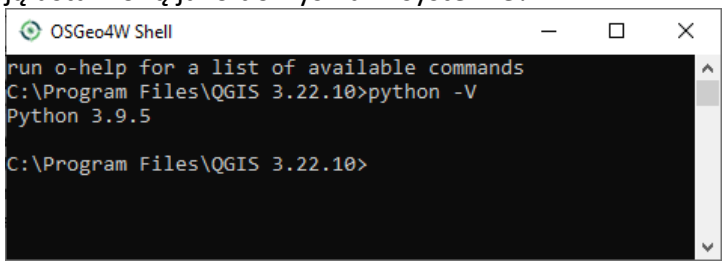
Uruchomienie konsoli umożliwia sprawdzenie wersji Python z jakiej obecnie korzystasz na swoim komputerze.



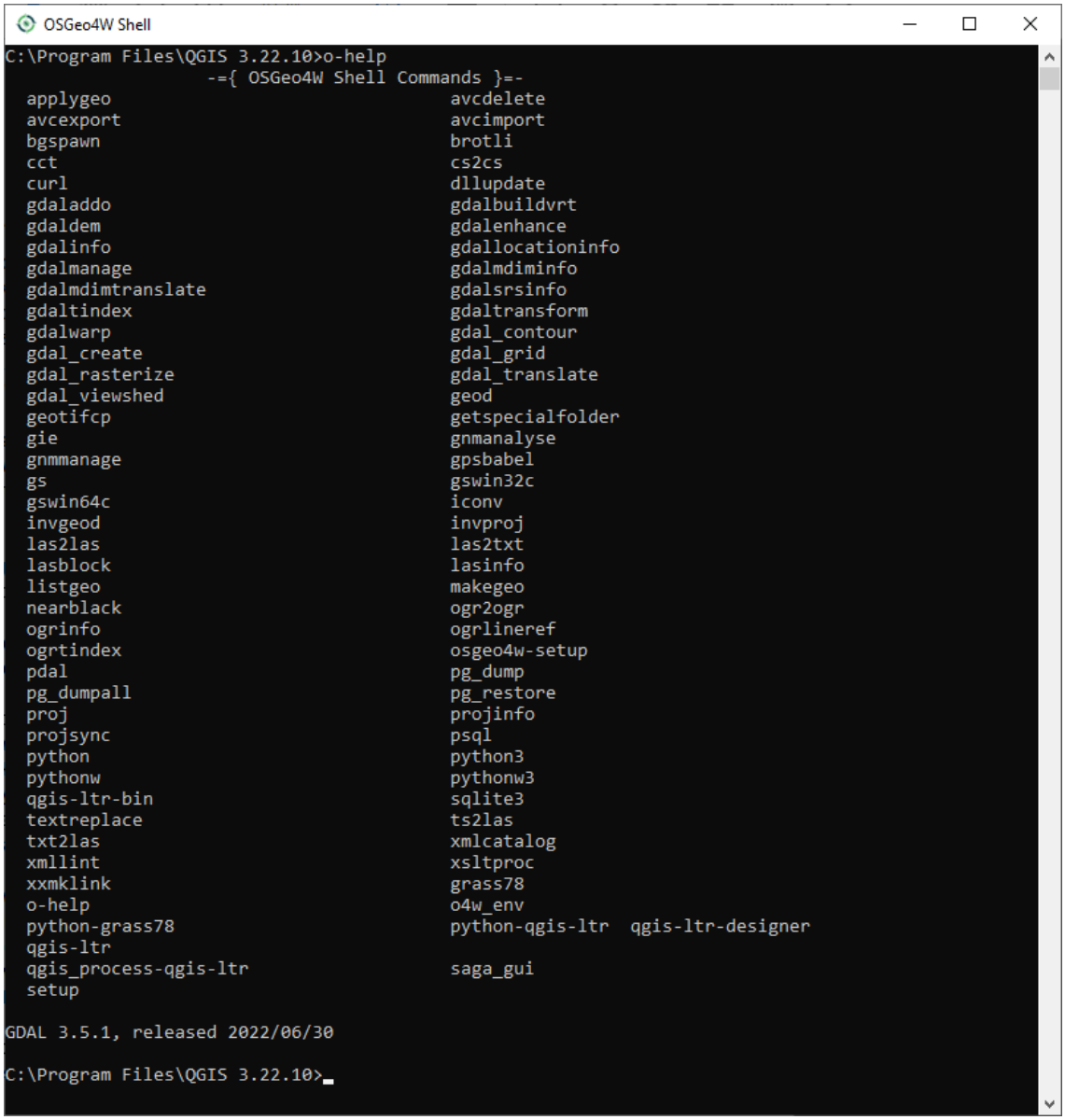
Aby sprawdzić dostępne polecenia można wpisać komendę o-help - powinny zostać wyświetlone podstawowe komendy QGIS.

Ćwiczenie 1. Sprawdzenie wersji Python i listy komend w oknie OSGeo4W Shell

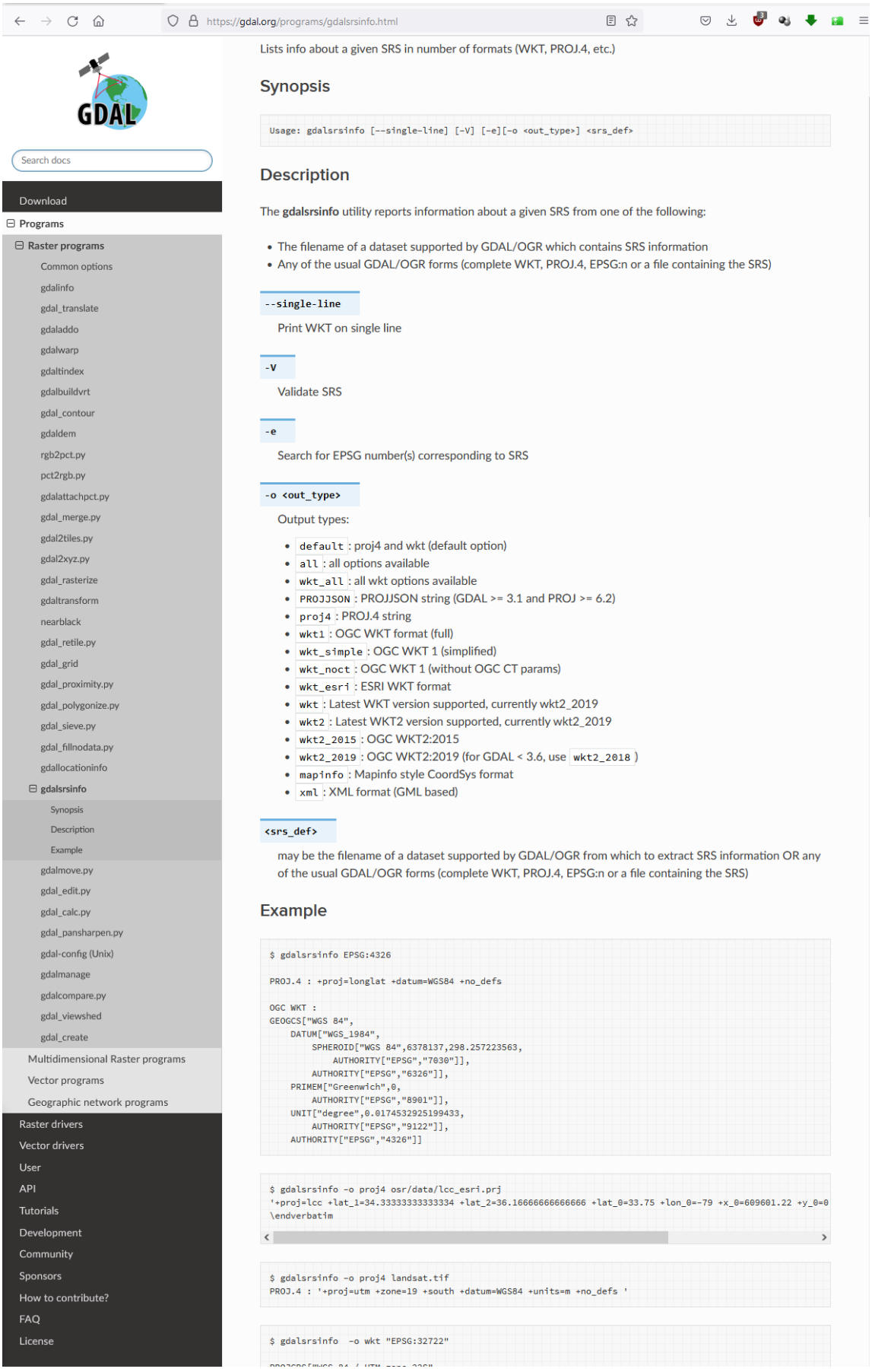
1. Uruchom okno OSGeo4W Shell
2. Wprowadź komendę **python -V** i potwierdź ją klikając Enter. Jaka wersja Pythona jest obecnie wersją ustawioną jako domyślna w systemie?



3. Wprowadź komendę o-help i zapoznaj się z lista poleceń jakie dostępne są w powłoce OSGeo4W



Informacje o poszczególnych poleceniach należy wyszukiwać po ich nazwie. Najczęściej projekty, np. dostarczające biblioteki/polecenia/algorytmy takie jak **GDAL** posiadają swoje strony www, gdzie przedstawiono zarówno opis działania, jak i składnię narzędzi.
Przykładowa strona www polecenia **gdalsrsinfo**: *https://gdal.org/programs/gdalsrsinfo.html*.



https://gdal.org/programs/gdalsrsinfo.html

GDAL

Search docs

- Download
- Programs
  - Raster programs
    - Common options
    - gdalinfo
    - gdal_translate
    - gdaladdo
    - gdalwarp
    - gdaltindex
    - gdalbuildvrt
    - gdal_contour
    - gdaldem
    - rgb2pct.py
    - pct2rgb.py
    - gdalattachpct.py
    - gdal_merge.py
    - gdal2tiles.py
    - gdal2xyz.py
    - gdal_rasterize
    - gdaltransform
    - nearblack
    - gdal_retile.py
    - gdal_grid
    - gdal_proximity.py
    - gdal_polygonize.py
    - gdal_sieve.py
    - gdal_fillnodata.py
    - gdallocationinfo
    - gdalsrsinfo
      - Synopsis
      - Description
      - Example
    - gdalmove.py
    - gdal_edit.py
    - gdal_calc.py
    - gdal_pansharpen.py
    - gdal-config (Unix)
    - gdalmanage
    - gdalcompare.py
    - gdal_viewshed
    - gdal_create
  - Multidimensional Raster programs
  - Vector programs
  - Geographic network programs
- Raster drivers
- Vector drivers
- User
- API
- Tutorials
- Development
- Community
- Sponsors
- How to contribute?
- FAQ
- License

Lists info about a given SRS in number of formats (WKT, PROJ.4, etc.)

Synopsis

```
Usage: gdalsrsinfo [--single-line] [-V] [-e][-o <out_type>] <srs_def>
```

Description

The **gdalsrsinfo** utility reports information about a given SRS from one of the following:

- The filename of a dataset supported by GDAL/OGR which contains SRS information
- Any of the usual GDAL/OGR forms (complete WKT, PROJ.4, EPSG:n or a file containing the SRS)

`--single-line`

Print WKT on single line

`-V`

Validate SRS

`-e`

Search for EPSG number(s) corresponding to SRS

`-o <out_type>`

Output types:

- `default` : proj4 and wkt (default option)
- `all` : all options available
- `wkt_all` : all wkt options available
- `PROJJSON` : PROJJSON string (GDAL >= 3.1 and PROJ >= 6.2)
- `proj4` : PROJ.4 string
- `wkt1` : OGC WKT format (full)
- `wkt_simple` : OGC WKT 1 (simplified)
- `wkt_noct` : OGC WKT 1 (without OGC CT params)
- `wkt_esri` : ESRI WKT format
- `wkt` : Latest WKT version supported, currently wkt2_2019
- `wkt2` : Latest WKT2 version supported, currently wkt2_2019
- `wkt2_2015` : OGC WKT2:2015
- `wkt2_2019` : OGC WKT2:2019 (for GDAL < 3.6, use `wkt2_2018`)
- `mapinfo` : Mapinfo style CoordSys format
- `xml` : XML format (GML based)

`<srs_def>`

may be the filename of a dataset supported by GDAL/OGR from which to extract SRS information OR any of the usual GDAL/OGR forms (complete WKT, PROJ.4, EPSG:n or a file containing the SRS)

Example

```
$ gdalsrsinfo EPSG:4326

PROJ.4 : +proj=longlat +datum=WGS84 +no_defs

OGC WKT :
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563,
            AUTHORITY["EPSG","7030"]],
        AUTHORITY["EPSG","6326"]],
    PRIMEM["Greenwich",0,
        AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.0174532925199433,
        AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4326"]]
```

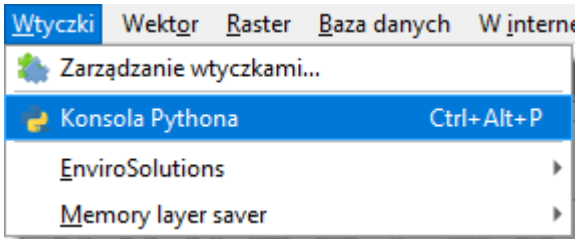
```
$ gdalsrsinfo -o proj4 osr/data/lcc_esri.prj
'+proj=lcc +lat_1=34.33333333333334 +lat_2=36.16666666666666 +lat_0=33.75 +lon_0=-79 +x_0=609601.22 +y_0=0
\endverbatim
```

```
$ gdalsrsinfo -o proj4 landsat.tif
PROJ.4 : '+proj=utm +zone=19 +south +datum=WGS84 +units=m +no_defs '
```

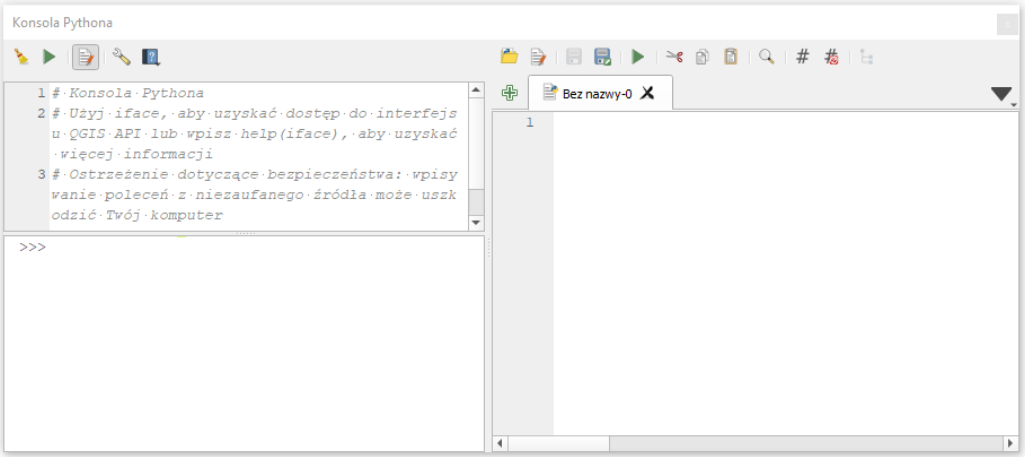
```
$ gdalsrsinfo  -o wkt "EPSG:32722"
```

Ćwiczenie 2. Uruchomienie konsoli Pythona w aplikacji QGIS – podstawowe elementy obsługi

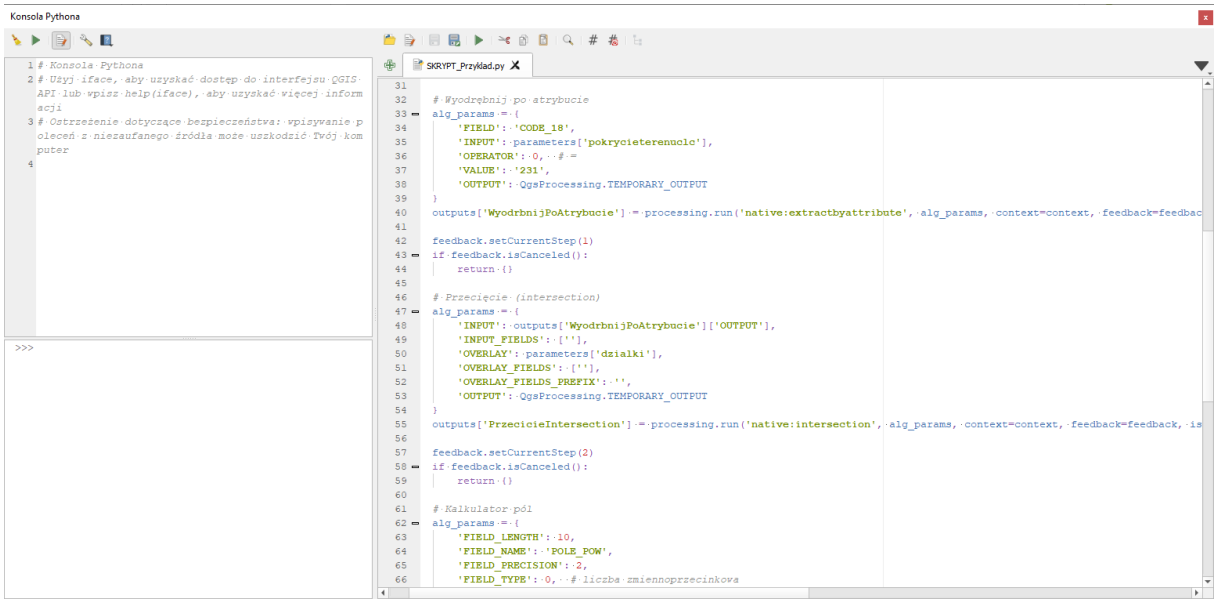
1. Otwórz aplikacje QGIS. W nowym oknie odszukaj ikonę konsoli Pythona , lub uruchom okno wybierając menu: Wtyczki→Konsola Pythona:



2. Zapoznaj się z oknem konsoli. Uruchom przyciskiem „Pokaż edytor” nowe okno w konsoli:



3. Za pomocą przycisku Wczytaj przykładowy plik ***SKRYPT_Przyklad.py*** zapisany w katalogu: **…DANE/KONSOLA_START/** - po wczytaniu przejrzyj jego strukturę. Wyszukaj znane Ci algorytmy i sprawdź, czy potrafisz rozpoznać ich ustawienia w kodzie Python:

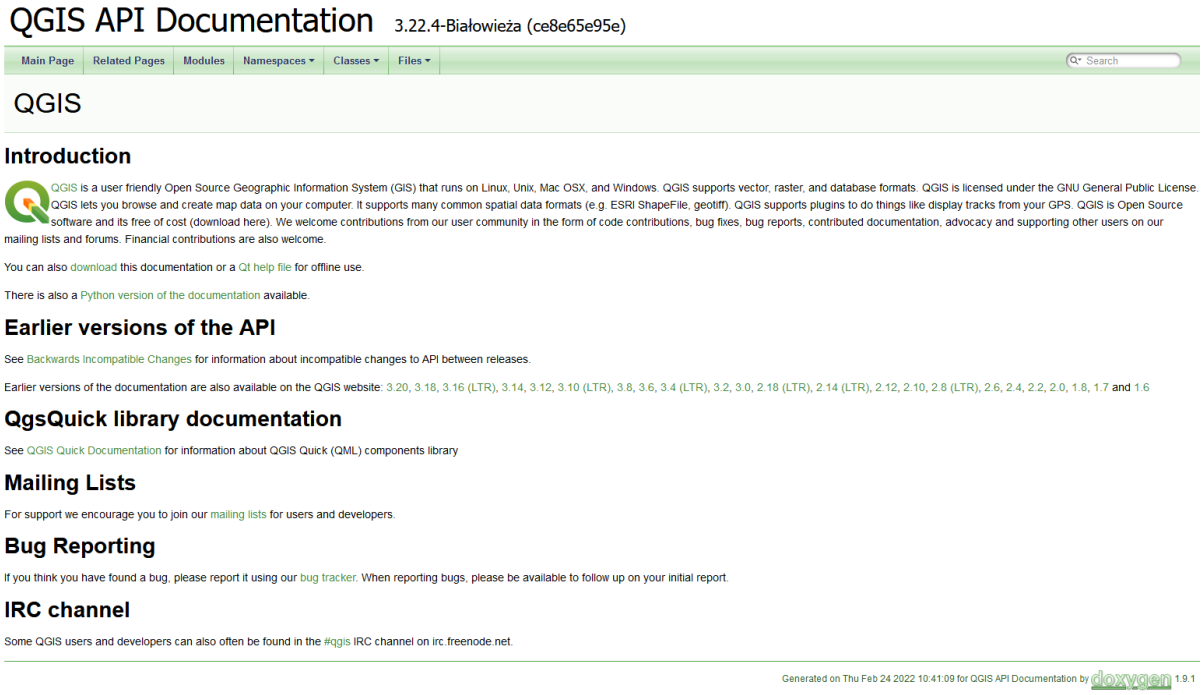


3. Biblioteki QGIS i ich wykorzystanie z poziomu języka Python

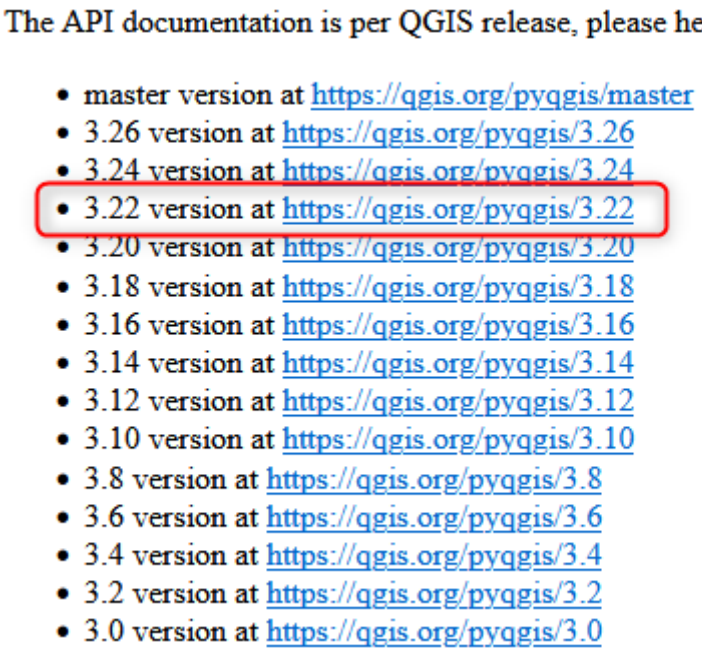
API, skrót od application programming interface czyli interfejs programowania aplikacji, jest to sposób komunikacji pomiędzy różnymi programami. Dzięki temu możliwe jest wydawanie poleceń z poziomu jednej aplikacji, które są wykonywane przez inny program. Przykładem mogą być zapytania HTTP(S) wysyłane 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



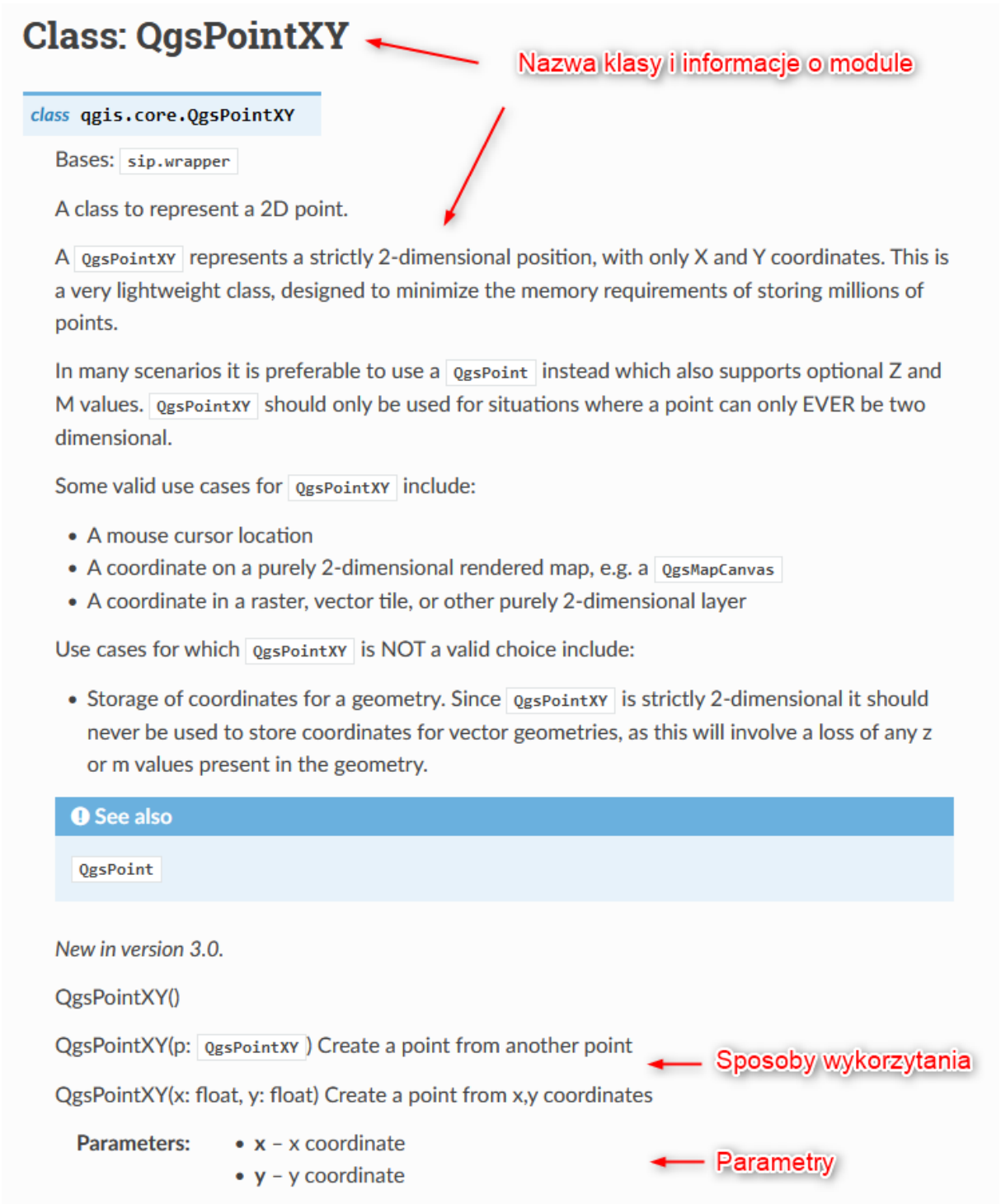
Źródło: https://api.qgis.org/api/3.22/



Źródło: https://api.qgis.org/api/

Strona główna (**https://www.qgis.org/pyqgis/3.22/**) zawiera wyszukiwarkę oraz spis wszystkich dostępnych klas z podziałem na moduły, w których się znajdują.

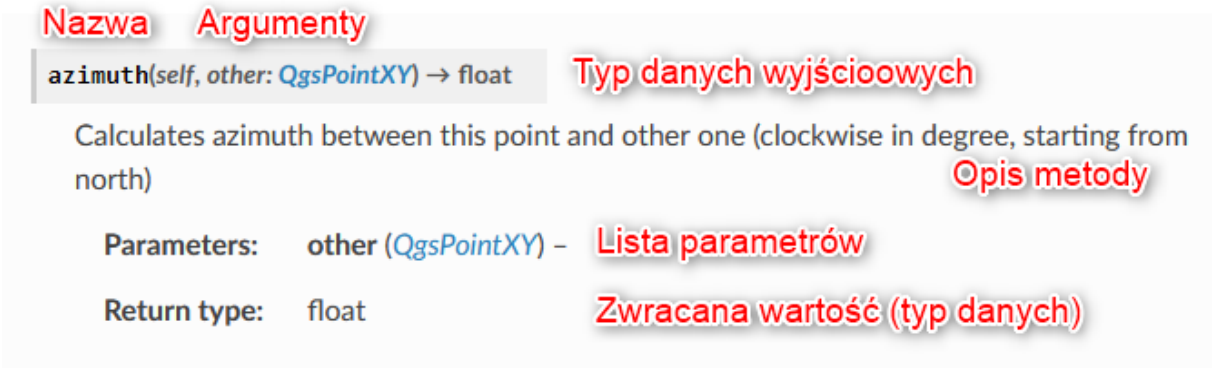
Po wejściu w dokumentację danej klasy znajdziemy w niej krótki jej opis oraz spis metod (funkcji) i atrybutów (zmiennych) dostępnych z jej poziomu. Po kliknięciu w nazwę można przejść do szczegółowszych informacji dotyczących danego elementu klasy np. jakie argumenty przyjmuje metoda i co zwraca jako wynik działania. Przykład opisu klasy w dokumentacji QGIS API dla języka Python. QgsPointXY – klasa reprezentująca punkt w przestrzeni 2D:



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.

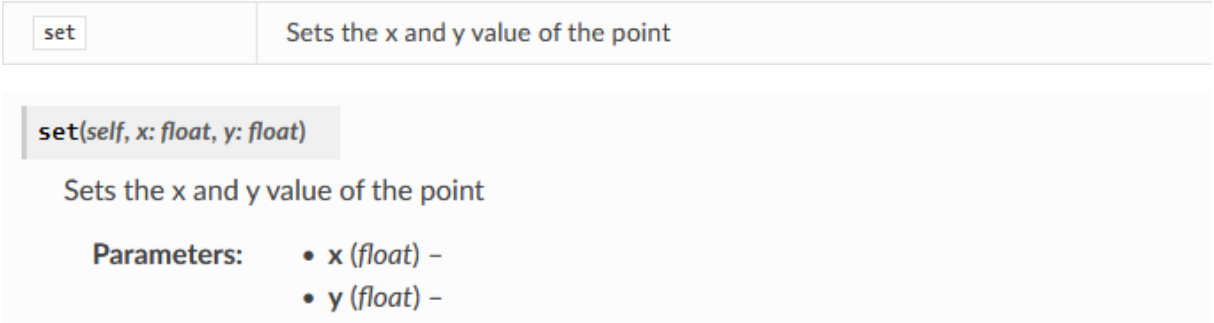
```
1. # Import klasy
2. from qgis.core import QgsPointXY
3. # Stworzenie instancji na podstawie współrzędnych
4. punkt1 = QgsPointXY( 10, 15 )
5. # Stworzenie instancji na podstawie istniejącej instancji tej samej klasy (klonowanie), punkt2 będzie miał te same współrzędne co punkt1
6. punkt2 = QgsPointXY( punkt1 )
```

Opis metod w dokumentacji:



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.

Przykład:




```
1. # Ustawienie współrzędnej X
2. punkt1.setX( 20.5 )
3. print( punkt1.x() )
4. # 20.5
5. print( punkt2.x() )
6. # 10
```

Biblioteki - import

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:

```
1. 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

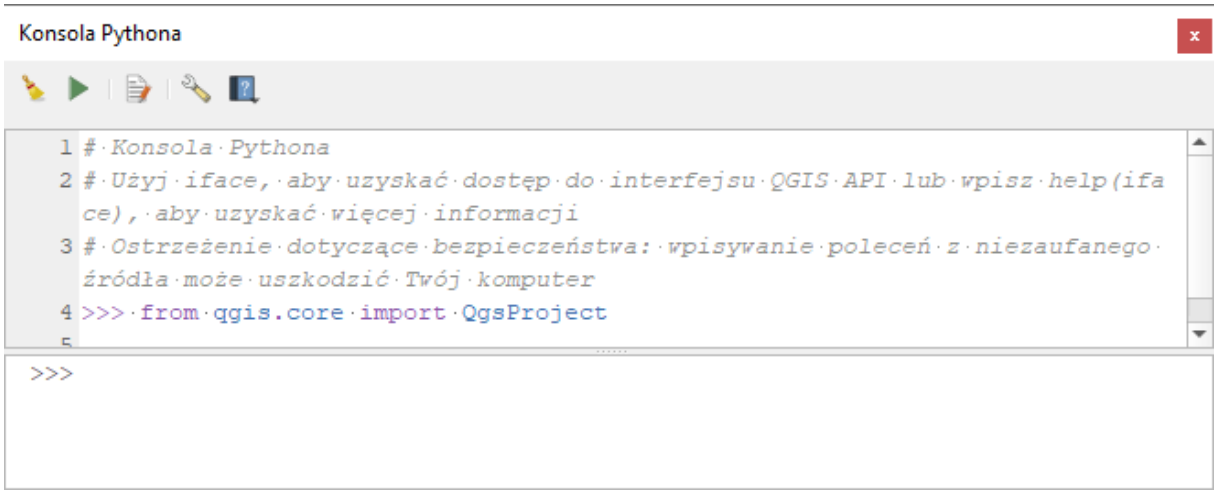
qgis.gui

- **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 3. Import klas w konsoli Pythona QGIS

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

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



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:

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

Wybrane metody obiektu iface:

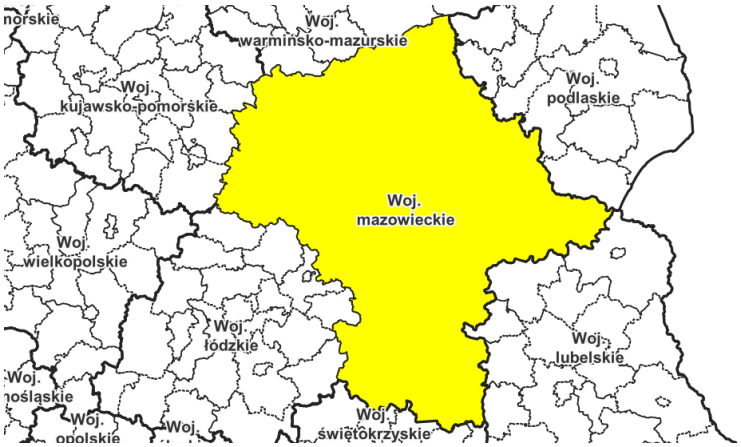
```
# Zwraca aktywną warstwę w QGIS
iface.activeLayer()
# <QgsMapLayer: 'nazwa' (ogr)>
# zwraca okno mapy QGIS (QgsMapCanvas)
iface.mapCanvas()
# <qgis._gui.QgsMapCanvas object at 0x...>
```

Ćwiczenie 4. Zasięg widocznej mapy

1. Otwórz projekt **ZASIEG.qgz** z folderu: **…\DANE\MAPA_ZASIEG**, a następnie powiększ widok mapy do wybranego województwa.
2. Uruchom Konsole Pythona, a następnie korzystając z metody mapCanvas wyświetl zasięg widoku mapy prezentowany przez współrzędne.

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.

```
1. iface.mapCanvas().extent()
2. <QgsRectangle: 379188.46878172358265147 345605.42331550002563745,
   919875.77110427641309798 634112.37723250000271946>
```

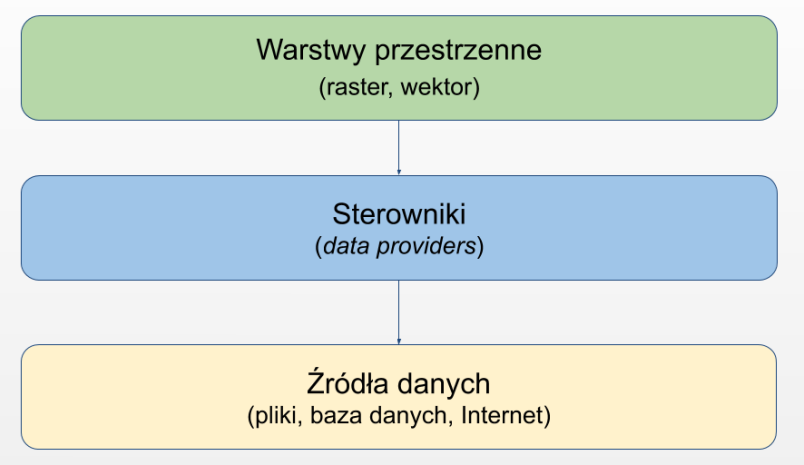


3. Sprawdź wynik, operację możesz powtórzyć dla innego województwa lub dla całego zasięgu danych.

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:

```
1. warstwa = iface.activeLayer()
2. print( type(warstwa) )
3. # <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**().

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

Obsługa danych rastrowych

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:

```
1. raster.width()
2. # 44
3. raster.unitsPerPixelX()
4. # 0.25
5. raster.bandCount()
6. # 36
7. raster.dataProvider()
8. # <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

```
1. raster.dataProvider().sourceNoDataValue(1)
2. # -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... }
```

Ćwiczenie 5. Kanały i ich wartości

1. Dodaj do widoku dane – pliki zawierające raster oraz obiekty punktowe - z folderu RASTER_WARTOSCI.
2. Stwórz w module analysis.py funkcję o nazwie raster_identify , która przyjmie jako argumenty warstwę rastrową (raster , instancja klasy QgsRasterLayer) oraz punkt (point , instancja klasy QgsPointXY) i zwróci dwie listy, pierwsza zawierająca numery kanałów, a druga odpowiadające im wartości rastra.

Na początku należy utworzyć definicję funkcji, która przyjmuje argumenty raster i point. Do odczytu wartości rastra w podanym punkcie należy wykorzystać metodę identify dostępną z poziomu sterownika warstwy rastrowej. Funkcja identify zwraca klasę

QgsRasterIdentifyResult , aby dostać się bezpośrednio do odczytanych danych należy wywołać metodę **results**(). Zwraca ona słownik, którego kluczem jest numer kanału a wartością odczytana wartość rastra. Należy teraz przekształcić ten słownik na dwie listy - pierwsza z numerami kanałów, druga z wartościami. W tym celu należy przygotować dwie puste listy i wypełnić je w pętli po elementach słownika. Aby mieć pewność, że elementy będą zwrócone w rosnącej kolejności kanałów, klucze słownika można posortować funkcją **sorted** . Na koniec należy zwrócić obiekt listy jako wynik działania funkcji.

```
1.  #Import użytej klasy
2.  from qgis.core import QgsRaster
3.  def raster_identify ( raster, point ):
4.  # Sterownik warstwy rastrowej (QgsRasterDataProvider)
5.  provider = raster.dataProvider()
6.  # Odczyt wartości komórek w podanym punkcie
7.  data = provider.identify( point, QgsRaster.IdentifyFormatValue )
8.  # Pobranie słownika z wartościami
9.  result = data.results()
10. # Lista kanałów rastra
11. bands = []
12. # Lista wartości rastra
13. values = []
14. #Iteracja po kanałach, sortujemy klucze słownika, aby mieć pewność,
15. # że zostaną zwrócone w rosnącej kolejności 1, 2, 3, itd.
16. for band in sorted(result):
17. # Dodanie kanału do listy
18. bands.append( band )
19. # Dodanie wartości do listy
20. values.append( result[band] )
21. # Zwrócenie obu list
22. return bands, values
```

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

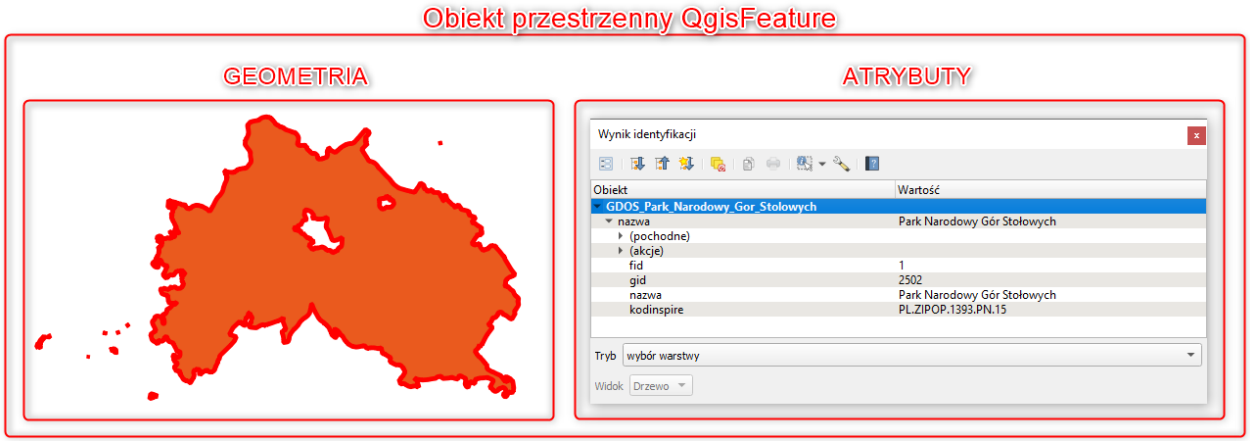
Przykłady użycia, *wektor* – instancja klasy **QgsVectorLayer**:

```
1.  wektor.featuresCount()
2.  # 16
3.  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]
```

Geometria

Główną klasą geometryczną jest **QgsGeometry**. Stanowi ona kontener dla geometrii dowolnego typu, możliwe jest przechowywanie m.in. punktów, linii, poligonów, geometrii wieloczęściowych (**multipart**), krzywych oraz różnych ich wariantów uwzględniających współrzędne Z i M. Geometria składa się z wierzchołków reprezentowanych przez klasę QgsPointXY (współrzędne XY) lub **QgsPoint** (współrzędne XYZM), które w zależności od typu są reprezentowane w formie list.

punkt (QgsPoint)	QgsPointXY(x, y)	
punkt wieloczęściowy (QgsMultiPoint)	[QgsPointXY(x1,y1), ..., QgsPointXY(xn, yn)]	
linia pojedyncza (QgsLineString)	[QgsPointXY(x1,y1), ..., QgsPointXY(xn, yn)]	
linia wieloczęściowa (QgsMultiLineString)	[[QgsPointXY(x1,y1), ..., QgsPointXY(xm, ym)], ... , [QgsPointXY(xn,yn), ..., QgsPointXY(xz, yz)]]	
prosty poligon (QgsPolygon)	[[QgsPointXY(x1,y1), ..., QgsPointXY(xn, yn)]]	
poligon z pierścieniem (QgsPolygon)	[[QgsPointXY(x1,y1), ..., QgsPointXY(xn, yn)], ... , [QgsPointXY(xm,ym), ..., QgsPointXY(xz, yz)]]	
poligon wieloczęściowy (QgsMultiPolygon)	[[[QgsPointXY(x1,y1), ..., QgsPointXY(xn, yn)], ... , [QgsPointXY(xm,ym), ..., QgsPointXY(xo, yo)]], ..., [[QgsPointXY(xp,yp), ..., QgsPointXY(xq, yq)]]]	

Należy podkreślić, że sama geometria reprezentuje dane w układzie kartezjańskim. Nie posiada żadnych informacji dodatkowych związanych z układem współrzędnych i porównując ze sobą obiekty tego typu należy zawsze pamiętać aby były w tym samym układzie

współrzędnych. Klasa QgsGeometry posiada bogaty zbiór metod służących do tworzenia, modyfikowania i zarządzania geometriami, które można podzielić na kilka kategorii.

Nowe geometrie można tworzyć na kilka sposobów:

- z listy wierzchołków (zgodnie z powyższą tabelą) - służą do tego metody rozpoczynające się od przedrostka from , które są wywoływane bezpośrednio z klasy **QgsGeometry**:

```
# Pojedynczy punkt
punkt = QgsGeometry.fromPointXY( QgsPointXY( 10 , 21.5 ) )
```

```
# Pojedyncza linia
linia = QgsGeometry.fromPolylineXY([QgsPointXY( 0 , 10 ),QgsPointXY( 11 , 16 )])
```

```
# Wielopunkt
wielopunkt = QgsGeometry.fromMultiPointXY( [ QgsPointXY( 0 , 0 ), QgsPointXY( 1 , 1 )] )
```

- przekształcając istniejące obiekty - metody zwracające nową instancję klasy **QgsGeometry** :

```
# Buforowanie, należy podać odległość i poziom zaokrąglenia krzywych (im wyższa wartość tym więcej punktów jest generowanych na łukach lub końcach linii)
bufor = geometria.buffer( 1000 , 5 )
```

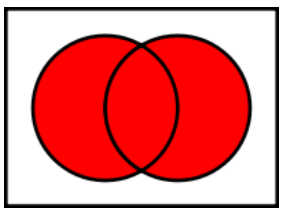
```
# Centroid (środek masy), może znajdować się poza geometrią źródłową
centroid = geometria.centroid()
```

```
# Punkt wewnątrz geometrii
punkt = geometria.pointOnSurface()
```

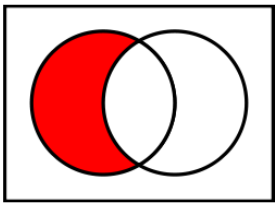
```
# Otoczka wypukła
poligon = geometria.convexHull()
```

- wykonanie jednej z operacji przestrzennych na dwóch geometriach:

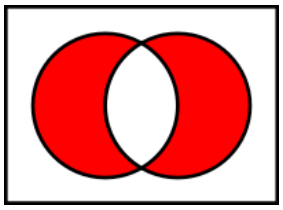
combine - połączenie dwóch geometrii w jedną,



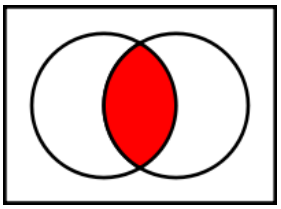
difference - zwraca geometrię, która nie jest wspólna z drugą,



symDifference - zwraca części rozłączne dla obu obiektów,



intersection - zwraca część wspólną obu geometrii,



nearestPoint - punkt na geometrii, leżący najbliżej innej geometrii.

Wszystkie metody wywołuje się podając jako parametr drugą geometrię:

```
1. nowa_geometria = geometria_1.combine( geometria_2 )
```

Informacje o geometrii

Metody, które zwracają informacje o danej geometrii:

```
1. # Sprawdzenie czy obiekt jest wieloczęściowy, zwraca wartość logiczną
2. print( geometria.isMultipart() )
3. # False
```

```
4. # Obwód lub długość (linie i poligony)
5. print( geometria.length() )
6. # 62.23
```

```
7. # Pole powierzchni (poligony)
8. print( geometria.area() )
9. # 6272.23
```

```
# Prostokąt ograniczający geometrię (najmniejsze i największe wartości
współrzędnych)
print( geometria.boundingBox() )
# <QgsRectangle: 517613.8493 352477.8594, 781450.3591 627244.5402>
```

Powierzchnia i długość są zwracane w jednostkach układu współrzędnych. Można również wyeksportować wierzchołki geometrii do listy za pomocą jednej z metod, których nazwa rozpoczyna się od przedrostka **as** :

```
# Pojedynczy punkt
print( geometria.asPoint() )
# QgsPointXY(10, 21.5)
```

```
# Pojedyncza linia
print( geometria.asPolyline() )
# [ QgsPointXY(0, 10), QgsPointXY(11, 16),... ]
```

```
# Punkt wieloczęściowy
print( geometry.asMultiPoint() )
# [ QgsPointXY(0, 0), QgsPointXY(1,1), ...]
```

Typ geometrii

QGIS definiuje dwa zbiory z typami geometrii. Są one zdefiniowane w klasie **QgsWkbTypes**.

- **GeometryType** - podział uproszczony na podstawowe typy geometrii (wartości z przyrostkiem Geometry):
  - PointGeometry ,
  - LineGeometry ,
  - PolygonGeometry ,
  - UnknownGeometry ,
  - NullGeometry.

Aby uzyskać informacje o tym typie należy wywołać metodę type() :

```
# Sprawdzenie czy dana geometria jest punktowa
print( geometria.type() == QgsWkbTypes.PointGeometry )
# True
```

Type - podział szczegółowy, zawiera wszystkie wspierane typy geometrii z podziałem na jedno- i wieloczęściowe, geometrie 2D, 25D, 3D, krzywe np. Point, MultiPoint, Point3D, PointZM itd. Atrybuty z tej grupy służą również do określania rodzaju geometrii przy tworzeniu nowych warstw wektorowych. Aby uzyskać informacje o tym typie należy wywołać metodę **wkbType**():

```
print( geometria.wkbType() == QgsWkbTypes.Point )
# True
print( geometria.wkbType() == QgsWkbTypes.MultiPoint )
# False
```

Zapytania przestrzenne

Zapytania przestrzenne badają wzajemne relacje pomiędzy obiektami geometrycznymi. Testują one konkretne przypadki relacji i najczęściej zwracają prawdę lub fałsz logiczny w zależności od wyniku. Najpopularniejsze zapytania (ich nazwy są również nazwami metod w klasie **QgsGeometry**):

- **intersects** - przecinanie, prawda jeśli geometrie mają część wspólną
- **contains** - zawieranie się, prawda jeśli jedna geometria w całości znajduje się w drugiej
- **touches** - stykanie, prawda gdy geometrie mają część wspólną ale nie pokrywają się częścią wewnętrzną (poligony)
- **overlaps** - nachodzenie, prawda jeśli obie geometrie nakładają się na siebie
- **crosses** - przecinanie, prawda jeśli jedna geometria dzieli drugą na dwie części
- **equals** - tożsamość, prawda jeśli dwie geometrie są takie same
- **disjont** - rozbieżność, prawda jeśli geometrie nie mają wspólnych fragmentów

Ćwiczenie 6. Wierzchołki obiektu poligonowego

1. Do nowego pustego okna mapy dodaj warstwę **park_narodowy.shp** z folderu: **…\DANE\ANALIZY_WIERZCHOLKI**.
2. W konsoli Pythona stwórz skrypt - Wykonaj pętlę na obiektach warstwy miasta i wydrukuj wierzchołki tworzące geometrię tych

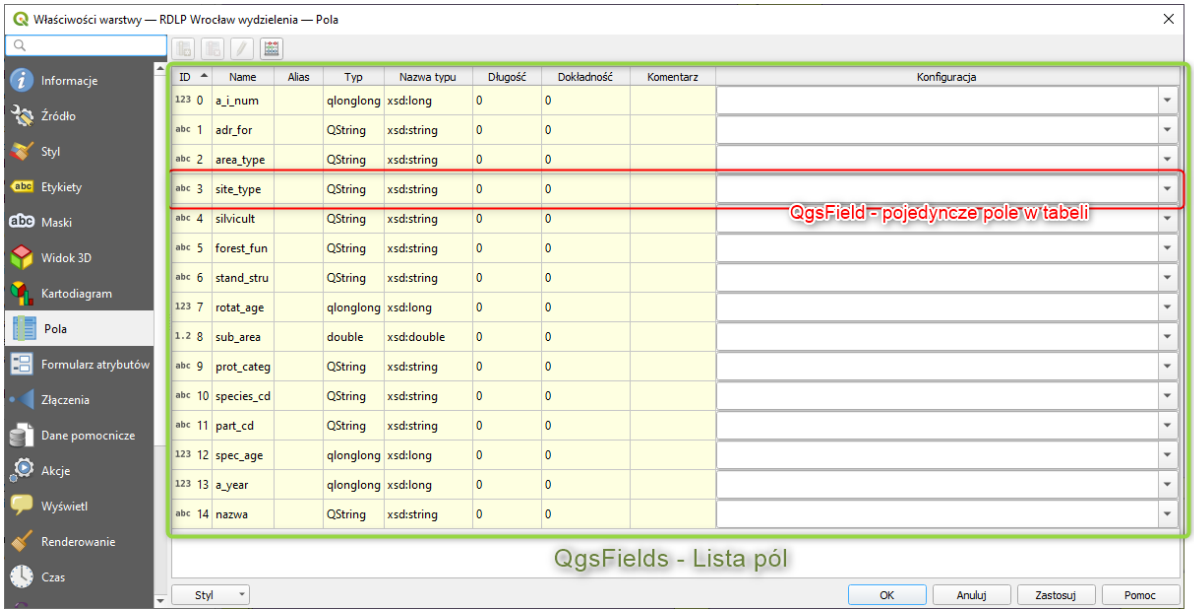
Aby uzyskać dostęp do warstwy **park_narodowy.shp** należy ją zaznaczyć w panelu legendy i wywołać w konsoli polecenie **warstwa**=iface.**activeLayer**() . W ten sposób pod zmienną warstwa mamy instancję klasy **QgsVectorLayer** . Kolejnym krokiem jest wykonanie pętli for po obiektach tej warstwy za pomocą metody **getFeatures**(). W każdej iteracji tej pętli otrzymamy dostęp do pojedynczego obiektu przestrzennego - instancji klasy **QgsFeature**. Klasa ta posiada metodę **geometry**(), która zwraca instancję klasy **QgsGeometry** reprezentującą geometrię podanego obiektu. Każda geometria składa się z wierzchołków, żeby je zwrócić należy wywołać, w zależności od typu geometrii, odpowiednią metodą. Warstwa park_narodowy.shp ma być punktowa więc właściwą metodą jest **asPoint**(). Wynik działania tej metody należy wydrukować funkcją **print**().

```
# Pobranie aktywnej warstwy
warstwa = iface.activeLayer()
```

```
3. #Iteracja po obiektach przestrzennych warstwy
4. for obiekt in warstwa.getFeatures():
5. # Pobranie geometrii obiektu
6. geometria = obiekt.geometry()
7. # Wyciągnięcie wierzchołków geometrii i ich wydrukowanie
8. print( geometria.asPoint() )
```

Tabela atrybutów

Tabelę atrybutów opisują dwie klasy **QGIS API**. **QgsField** opisuje pojedynczą kolumnę m.in. jej nazwę, typ danych, długość. Natomiast **QgsFields** to klasa, w której przechowywane są informacje o kolumnach i ich kolejności, jest kontenerem dla instancji klas **QgsField** .



Warto podkreślić, że **w QGIS kolejność kolumn ma znaczenie**. Każda kolumna ma przypisany indeks określający jej kolejność na liście. Na powyższym obrazku jest to kolumna ID. Większość metod, w których określa się kolumnę wymaga podania jej indeksu. Niektóre pozwalają podać również nazwę.

QgsFields

Główne metody klasy służące do pobrania informacji o tabeli:

```
1. # Liczba kolumn
2. tabela.count()
```

```
1. # Lista nazw kolumn
2. tabela.names()
3. # [ 'ID', 'Opis', "Powierzchnia' ]
```

```
# Indeks kolumny o podanej nazwie, zwróci -1 jeśli nie znaleziono
kolumny
tabela.indexFromName( 'Opis' )
```

```
# Pobranie pojedynczego pola (QgsField), można podać nazwę lub indeks
kolumny
tabela.field( 0 )
# <QgsField: ID (Int)>
tabela.field( 'Opis' )
# <QgsField: Opis (String)>
```

QgsField

Z poziomu tej klasy można uzyskać informacje o konkretnej kolumnie:

```
# nazwa pola
kolumna.name()
# Opis
```

```
# długość pola (ilość znaków tekstu lub cyfr w liczbach)
kolumna.length()
```

```
# dokładność liczb rzeczywistych (ilość cyfr po przecinku)
kolumna.precision()
```

```
# typ pola, zwracany jako wartość z klasy QVariant (została ona
szczegółowo omówiona w części "Zarządzanie schematem tabeli atrybutów")
kolumna.type() == QVariant.String
# True
```

Zarządzanie schematem tabeli atrybutów

Stworzenie nowego schematu atrybutów:

```
# Pusta tabela (bez kolumn)
tabela = QgsFields()
```

```
# Skopiowanie schematu z innej tabeli
tabela = QgsFields( inna_tabela )
```

Rozszerzenie istniejącego schematu o pola z innej tabeli. Kolumny zostaną dodane na końcu schematu zgodnie z oryginalną kolejnością:

```
tabela.extend( inna_tabela )
```

Dodanie pola do tabeli:

```
1. # Import klasy do określenia typu danych
2. from qgis.PyQt.QtCore import QVariant
```

```
1. # Utworzenie nowej kolumny
2. kolumna_opis = QgsField( 'opis' ,QVariant.String)
```

```
1. # Dodanie kolumny do tabeli
2. tabela.append( kolumna_opis )
```

Klasa **QVariant** pochodzi z frameworka Qt . W QGIS jest ona używana do zdefiniowania typu danych dla danego pola tabeli atrybutów.

Typy danych wykorzystywane przez QGIS:

- **QVariant.String** - łańcuch znaków
- **QVariant.Int** - liczby całkowite
- **QVariant.Double** - liczby rzeczywiste
- **QVariant.Date** , **QVariant.DateTime** - data oraz data z czasem

Usunięcie kolumny odbywa się poprzez podanie jej indeksu:

```
1. tabela.remove( 0 )
```

Edycja

Stworzenie obiektu przestrzennego

Tworząc nowy obiekt przestrzenny **QgsFeature** należy podać schemat tabeli atrybutów (**QgsFields**), a następnie przypisać mu geometrię i wartości atrybutów:

```
1. # Stworzenie obiektu przestrzennego
2. obiekt = QgsFeature( tabela )
3. # Stworzenie geometrii
4. geometria = QgsGeometry.fromPointXY( QgsPointXY( 20 , 51 ) )
5. # Przypisanie geometrii do obiektu
6. obiekt.setGeometry( geometria )
7. # Przypisanie pojedynczej wartości
8. obiekt[ 'opis' ] = 'Obiekt przestrzenny'
9. #Przypisanie wielu wartości jednocześnie
```

```
10. obiekt.setAttributes( [ 20 , 'Opis' ] )
```

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**():

```
1. 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.

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

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

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

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

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

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

```
1. atrybuty = {
2. 0 : 20 , # Pierwsze pole
3. 1 : 'opis' # Drugie pole
4. }
5. # Zmiana wartości dwóch pól w obiekcie o ID 1
6. 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 ] )
```

Ćwiczenie 7. Dodawanie pola z powierzchnią obiektów

1. Do nowego widoku dodaj warstwę **działki.shp** z folderu **..DANE/OBSLUGA_TABELI**, a następnie uruchom konsolę Python.
2. Celem ćwiczenia jest dodanie nowego pola area przechowującego liczby rzeczywiste i wypełnienie go wartościami reprezentującymi powierzchnię każdego poligonu wyrażoną w km^2 .

Aby uzyskać dostęp do warstwy miasta należy ją zaznaczyć w panelu legendy i wywołać w konsoli polecenie **warstwa**=**iface**.**activeLayer**() . W ten sposób pod zmienną warstwą mamy instancję klasy **QgsVectorLayer** . Można również wyciągnąć sterownik tej warstwy do osobnej zmiennej, ponieważ będzie potrzebny w kilku miejscach. Kolejnym krokiem jest utworzenie nowej kolumny o podanej nazwie, czyli tworzona jest instancja klasy **QgsField** . Jako typ mają być przechowywane liczby rzeczywiste, więc należy podać typ **QVariant** . **Double** . Następnie za pomocą sterownika dodajemy nową kolumnę do źródła danych i odświeżamy warstwę w QGIS metodą **reload**() . 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. Mając instancję klasy **QgsFeature** pobieramy identyfikator (**metoda id**()), a z geometrii tego obiektu wyliczamy jej powierzchnię (**metoda area**()). 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 liczba określająca powierzchnię. Na końcu możemy odświeżyć informacje o zmianach w źródle danych za pomocą metody **reload**() .

```
1.  # Pobranie aktywnej warstwy
2.  warstwa = iface.activeLayer()
3.  # Sterownik warstwy wektorowej
4.  sterownik = warstwa.dataProvider()
5.  # Utworznie nowego pola
6.  pole = QgsField( 'area' , QVariant.Double )
7.  # Zapisanie pola w źródle danych
8.  sterownik.addAttributes( [pole] )
9.  # Odświeżenie informacji w QGIS odnośnie zmian w źródle danych
10. warstwa.reload()
11. # Pobranie indeksu nowego pola
12. indeks = warstwa.fields().indexFromName( 'area' )
13. # Iteracja po obiektach przestrzennych warstwy
14. for obiekt in warstwa.getFeatures():
15. # Pobranie geometrii
16. geometria = obiekt.geometry()
17. # Zapisanie wartości w źródle danych, podajemy zagnieżdżone słowniki,
18. # kluczem pierwszego jest identyfikator obiektu, który będzie  modyfikowany,
19. # kluczem drugie indeks pola do aktualizacji
20. sterownik.changeAttributeValues ({
21. obiekt.id(): { indeks : geometria.area()/ 1000 }
22. })
23. # Odświeżenie informacji w QGIS odnośnie zmian w źródle danych
24. warstwa.reload()
```

4. Wykonywanie analiz przestrzennych z poziomu komend w języku Python

QGIS posiada bogaty zbiór narzędzi analitycznych zebrany w panelu Algorytmy Processingu. Główne możliwości, które daje panel algorytmów to:

- wykonywanie algorytmów pojedynczo lub w trybie wsadowym,
- budowanie modeli na bazie dostępnych algorytmów,
- algorytmy natywne oraz pochodzące z zewnętrznych aplikacji tj. GRASS GIS, SAGA GIS, Orfeo Toolbox, GDAL,
- tworzenie własnych skryptów (algorytmów) w języku Python,
- możliwość dodawania kolejnych algorytmów z pomocą wtyczek.

Dodatkowo QGIS posiada system rozszerzeń za pomocą wtyczek napisanych w języku Python. W porównaniu do algorytmów mają one większe możliwości jeżeli chodzi o integrację z QGIS (własne okna i panele) i mogą działać w tle, reagując na działania użytkownika. Za pomocą Menedżera wtyczek można instalować wtyczki ze specjalnych repozytoriów. Domyślne dostępne jest oficjalne repozytorium QGIS.

Wykorzystując język Python do wykonywania analiz w QGIS można korzystać z dodatkowych bibliotek, które muszą być zainstalowane w interpreterze tego języka, z którego korzysta QGIS. Istnieje wiele przydatnych bibliotek (część jest instalowana razem z QGIS), które ułatwiają skomplikowane obliczenia przestrzenne, statystyczne, czasowe czy wizualizację wyników. Najbardziej znane to:

- **NumPy** - dodaje nowe typy danych dla wielowymiarowych macierzy i tablic oraz funkcje matematyczne do ich analiz,
- **pandas** - manipulacja i analiza danych tabelarycznych i ciągów czasowych, pozwala importować dane z wielu różnych źródeł,
- **matplotlib** - tworzenie zaawansowanych wizualizacji w Python m.in. wykresy, grafy, oparta o tablice **NumPy** ,
- **SciPy** - bogaty zestaw narzędzi do analiz matematycznych, statystycznych, naukowych i technicznych, oparta o tablice **NumPy** , wykorzystuje również inne wymienione biblioteki.

Wykorzystywanie powyższych narzędzi (określanych czasem zbiorczo jako SciPy Stack lub NumPy Stack) rozszerza znacząco możliwości analityczne języka Python, dzięki czemu jest on coraz częściej wykorzystywany w Data Science i stawiany obok takich narzędzi jak język R, MATLAB, GNU Octave, Excel, Apache Spark czy Tableau.

WAŻNE - ĆWICZENIA – Organizacja pracy

Przygotowane ćwiczenia maja na celu usamodzielnienie uczestników szkolenia w pracy ze składnią Python. W ramach podpowiedzi w przygotowywaniu kodu Python należy korzystać

ze strony https://www.qgis.org/pyqgis/3.22/ , a także przygotowanych przykładowych rozwiązaniach dla ćwiczeń (folder: SKRYPTY).

Ćwiczenia będą wykonywane wraz z instruktorem wyjaśniającym poszczególne etapy składania kodu.

Ćwiczenie 8. Buforowanie

1. Wykonaj buforowanie warstwy cieków (folder …DANE/ANALIZY_BUFOROWANIE).

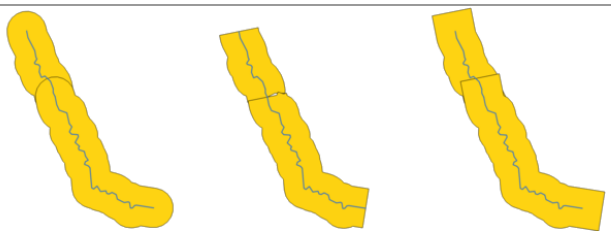
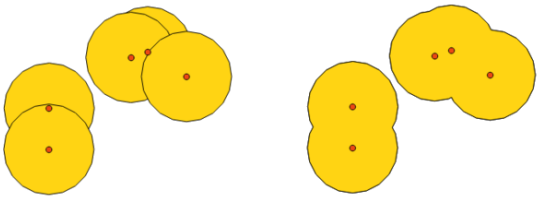
   Przyjmij dla działania skryptu następujące ustawienia:

   a. Odległość buforowania: 100m
   b. Segmenty: 5
   c. Zakończenie: zaokrąglony
   d. Styl połączenia: zaokrąglony
   e. Limit fazy: 2,00
   f. Agreguj wyniki: tak
   g. Zapis do warstwy tymczasowej

Skrypt: **buforowanie_100.py**

Przykładowy opis dla funkcji ze strony pomocy dla narzędzia:

Label	Name	Type	Description
Input layer	INPUT	[vector: any]	Input vector layer
Distance	DISTANCE	[number] Default: 10.0	Buffer distance (from the boundary of each feature). You can use the Data Defined button on the right to choose a field from which the radius will be calculated. This way you can have different radius for each feature (see Variable distance buffer).
Segments	SEGMENTS	[number] Default: 5	Controls the number of line segments to use to approximate a quarter circle when creating rounded offsets.
End cap style	END_CAP_STYLE	[enumeration] Default: 0	Controls how line endings are handled in the buffer. One of: • 0 — Round • 1 — Flat • 2 — Square

			 Fig. 25.53 Round, flat and square cap styles
Join style	JOIN_STYLE	[enumeration] Default: 0	Specifies whether round, miter or beveled joins should be used when offsetting corners in a line. Options are: • 0 — Round • 1 — Miter • 2 — Bevel  Fig. 25.54 Round, miter, and bevel join styles
Miter limit	MITER_LIMIT	[number] Default: 2.0	Controls the maximum distance from the offset curve to use when creating a mitered join (only applicable for miter join styles). Minimum: 1.
Dissolve result	DISSOLVE	[boolean] Default: False	Dissolve the final buffer. If True (checked), overlapping buffers will be dissolved (combined) into a new feature.  Fig. 25.55 Standard and dissolved buffer
Buffered	OUTPUT	[vector: polygon] Default: [Create temporary layer]	Specify the output (buffer) layer. One of: • Create Temporary Layer (TEMPORARY_OUTPUT) • Save to File… • Save to Geopackage… • Save to Database Table… • Append to Layer… The file encoding can also be changed here.

Outputs

Label	Name	Type	Description
Buffered	OUTPUT	[vector: polygon]	Output (buffer) polygon layer

Python code

Algorithm ID: native:buffer

```
1. import processing
2. processing.run("algorithm_id", {parameter_dictionary})
```

Ćwiczenie 9. Przycinanie danych do określonego zasięgu

1. Do nowego projektu dodaj warstwy z folderu: …DANE/ANALIZY_PRZYCINANIE (cieki oraz granica Parku Narodowego Gór Stołowych).
2. Wykorzystując konsolę Pythona skonfiguruj skrypt który wykonuje funkcję „Przytnij" dla warstwy cieków. Efektem ma być siec cieków na obszarze Parku Narodowego.

Skrypt: **przycinanie.py**

przycinanie.py

```
1  import processing
2
3  processing.run("native:clip", \
4  {'INPUT':'C:/DANE/ANALIZY_PRZYCINANIE/cieki.shp', \
5  'OVERLAY':'C:/DANE/ANALIZY_PRZYCINANIE/park_obszar.shp',\
6  'OUTPUT':'C:/DANE/ANALIZY_PRZYCINANIE/cieki_park.shp'})
7
8  iface.addVectorLayer('C:/DANE/ANALIZY_PRZYCINANIE/cieki_park.shp', '', 'ogr')
```

Ćwiczenie 10. Generowanie wierzchołków dla obiektów z warstwy poligonowej.

1. Do nowego projektu dodaj warstwę z folderu: **…DANE/ANALIZA_WIERZCHOLKI**. Warstwa przedstawia obszar powiatu kłodzkiego.
2. Uruchom konsolę Pythona i skonfiguruj skrypt który umożliwi pozyskanie wierzchołków z poligonu reprezentującego obszar powiatu kłodzkiego.

Skrypt: **wierzcholki.py**

Ćwiczenie 11. Agregacja danych PRG

1. Do nowego projektu dodaj dane PRG z folderu **…\DANE\ANALIZY_AGREGACJA**.
2. W konsoli Pythona, przy użyciu dostępnych pomocy, stwórz skrypt dodający nowe pole do warstwy powiatów i wyliczający kod województwa. Następnie taką warstwę poddaj agregacji po nowo dodanym polu w celu wygenerowania poligonów reprezentujących województwa.

Skrypt: **agregacja_prg.py**

Ćwiczenie 12. Podstawowe atrybuty geometrii w tabeli atrybutów

1. Do nowego okna dodaj warstwę reprezentującą działki ewidencyjne: ..\DANE\ANALIZY_ATRYBUTY_GEOM.
2. W konsoli Pythona stwórz skrypt, który doda nowe pola w tabeli warstwy zawierające podstawowe atrybuty geometrii (np.: powierzchnię, obwód, współrzędne obiektu lub centroidu obiektu.. itp.)

Skrypt: **podstawowe_atr_geom.py**

Ćwiczenie 13. Wsadowa zmiana układu współrzędnych

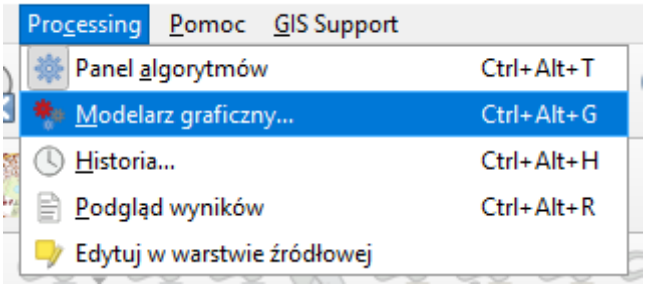
1. Otwórz nowy projekt QGIS, następnie w oknie przeglądarki przejdź do katalogu **…DANE/ ANALIZY_UKLAD_WSADOWO**. Zapoznaj się z jego zawartością. Sprawdź ile jest warstw we wskazanym katalogu.
2. Otwórz konsolę Pythona i stwórz skrypt który ma na celu przeliczenie wszystkich plików z danego katalogu do układu o kodzie **EPSG: 4326**. Nowo powstałe warstwy powinny mieć dopisane w nazwie „**_84**”.

Skrypt: **zmiana_ukladu_wsad.py**

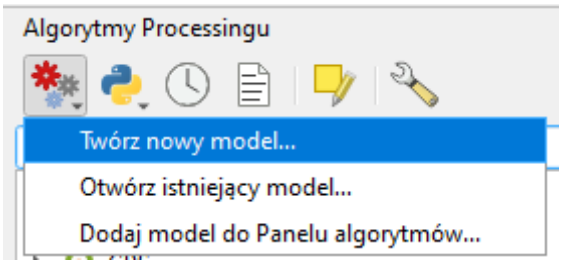
5. Tworzenie modelu przetwarzania danych w interfejsie graficznym QGIS – Modelarz

Modelarz graficzny umożliwia zapisanie kolejnych etapów analizy w postaci graficznej. Szczególnie przydatny może być w sytuacji kiedy musimy wielokrotnie powtarzać tą samą analizę np. dla różnych obszarów.

Okno modelarza otwierane jest z menu górnego **Processing → Modelarz graficzny**:



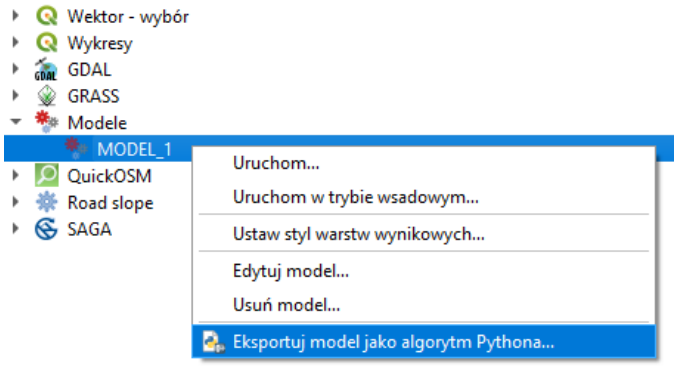
Można je również otworzyć z poziomu panelu **Algorytmy processingu**:

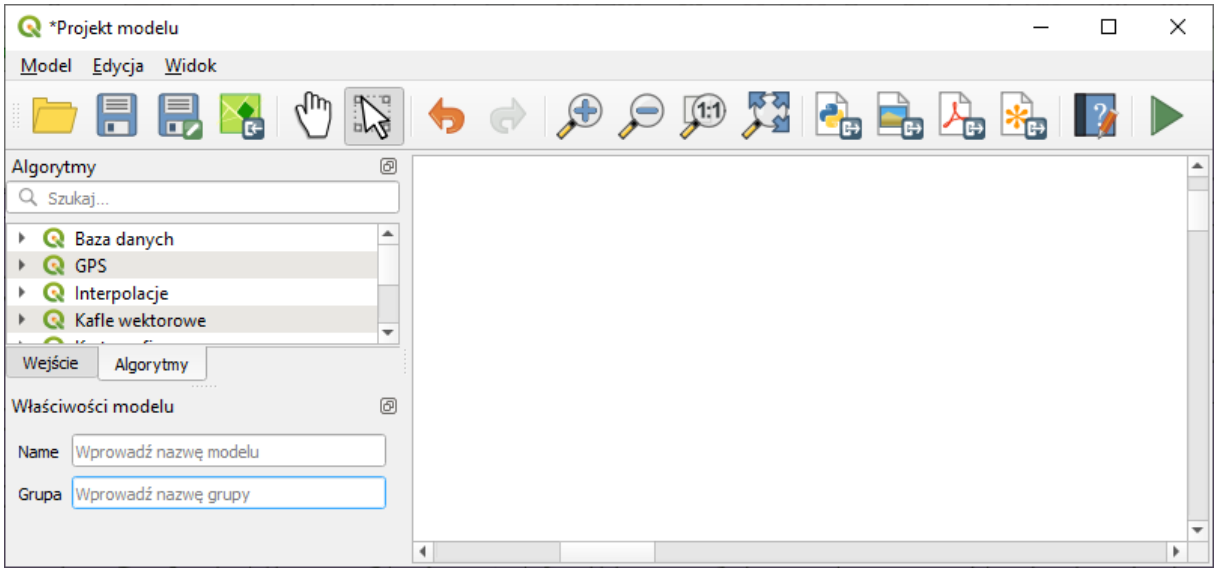


W oknie Modelarza znajdziemy m.in.:

- Okno modelu - obszar gdzie będziemy dodawać kolejne elementy,
- Panel Wejście – zawierający typy danych/parametrów wejściowych modelu,
- Panel Algorytmy – zawierający algorytmy processingu aplikacji,
- Właściwości modelu – panel, w którym ustalamy nazwę i grupę tworzonego modelu.

Jedną z ważniejszych funkcji związanych z tworzeniem modelu przetwarzania danych jest możliwość zapisu skonfigurowanego procesu do skryptu Pythona.





Ćwiczenie 14. Tworzenie modelu geoprzetwarzania w Modelarzu graficznym

Przygotujmy model, wyznaczający fragmenty działek, które mogłyby zostać wykorzystane jako tereny rekreacyjne. Przyjmijmy następujące założenia:

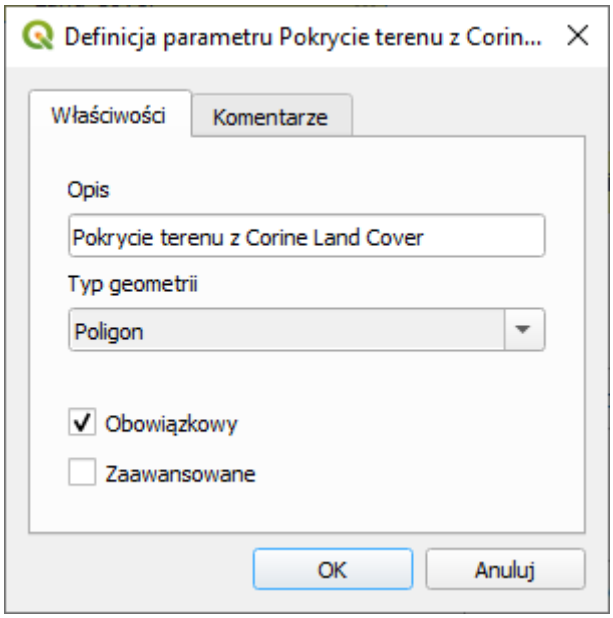
- Działki – ponieważ działki pochodzące z różnych powiatów mogą posiadać różną strukturę atrybutów ustalmy, że użytkownik musi wskazać warstwę zawierającą poligony z działkami dla sprawdzanego obszaru.
- Pokrycie terenu – wyznaczone działki muszą znajdować się na terenie łąk. Skorzystajmy z bazy pokrycia terenu Corin Land Cover 2018. Jest ona ustandaryzowana, zatem możemy w modelu zadać odpowiednie kryterium (łąki przypisany mają kod 231 - atrybut CODE_18).
- Pole powierzchni działki – ustawmy jako parametr określany i podawany przez użytkownika w arach. Przy czym załóżmy, że najmniejszy obszar nadający się do inwestycji musi mieć co najmniej 50 arów.

Wykorzystamy algorytmy:

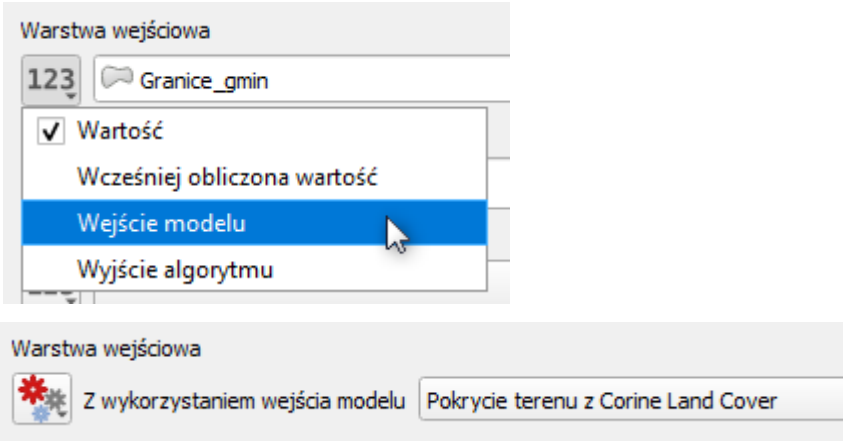
- **Wyodrębnij po atrybucie** – pozwoli nam wyznaczyć obszary łąk oraz obszary spełniające warunek minimalnej powierzchni.
- **Przecięcie** – wyznaczy fragmenty działek, które znajdują się na terenie łąk.
- **Kalkulator pół** – posłuży do obliczenia pola powierzchni fragmentów działek w arach.

Instrukcje

3. Do nowego projektu dodaj warstwy znajdujące się w geopaczce **Modelarz.gpkg** w katalogu **D:\DANE\MODEL**.
4. Powiększ się do zasięgu warstwy **Granice_gmin**.
5. Otwórz **Modelarz graficzny**.
6. W panelu **Wejście** po prawej stronie odszukaj **Warstwa wektorowa** i przeciągnij na okno modelu.
7. Podaj opis **Pokrycie terenu z Corine Land Cover**. Wskaż jako typ geometrii **poligon** i zaznacz parametr jako **obowiązkowy**:

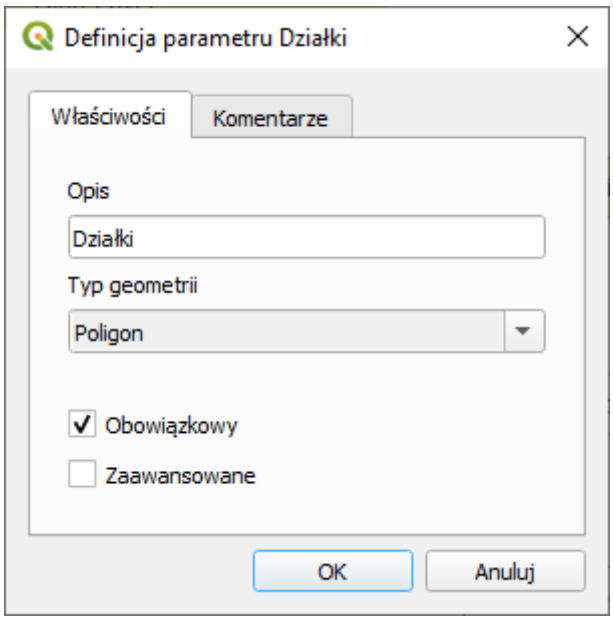


8. W panelu algorytmy odszukaj za pomocą wyszukiwarki funkcję **Wyodrębnij po atrybucie** i dodaj do modelu.
9. We właściwościach narzędzia ustaw:
    a. Warstwa wejściowa: Wejście modelu → Pokrycie terenu z CLC.

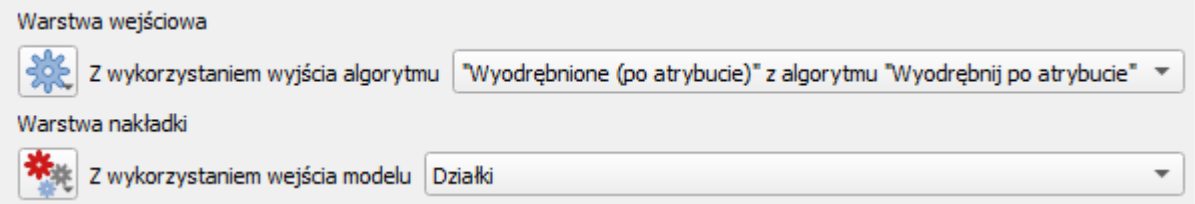


b. Atrybut zaznaczenia: CODE_18
c. Operator: =
d. Wartość: 231

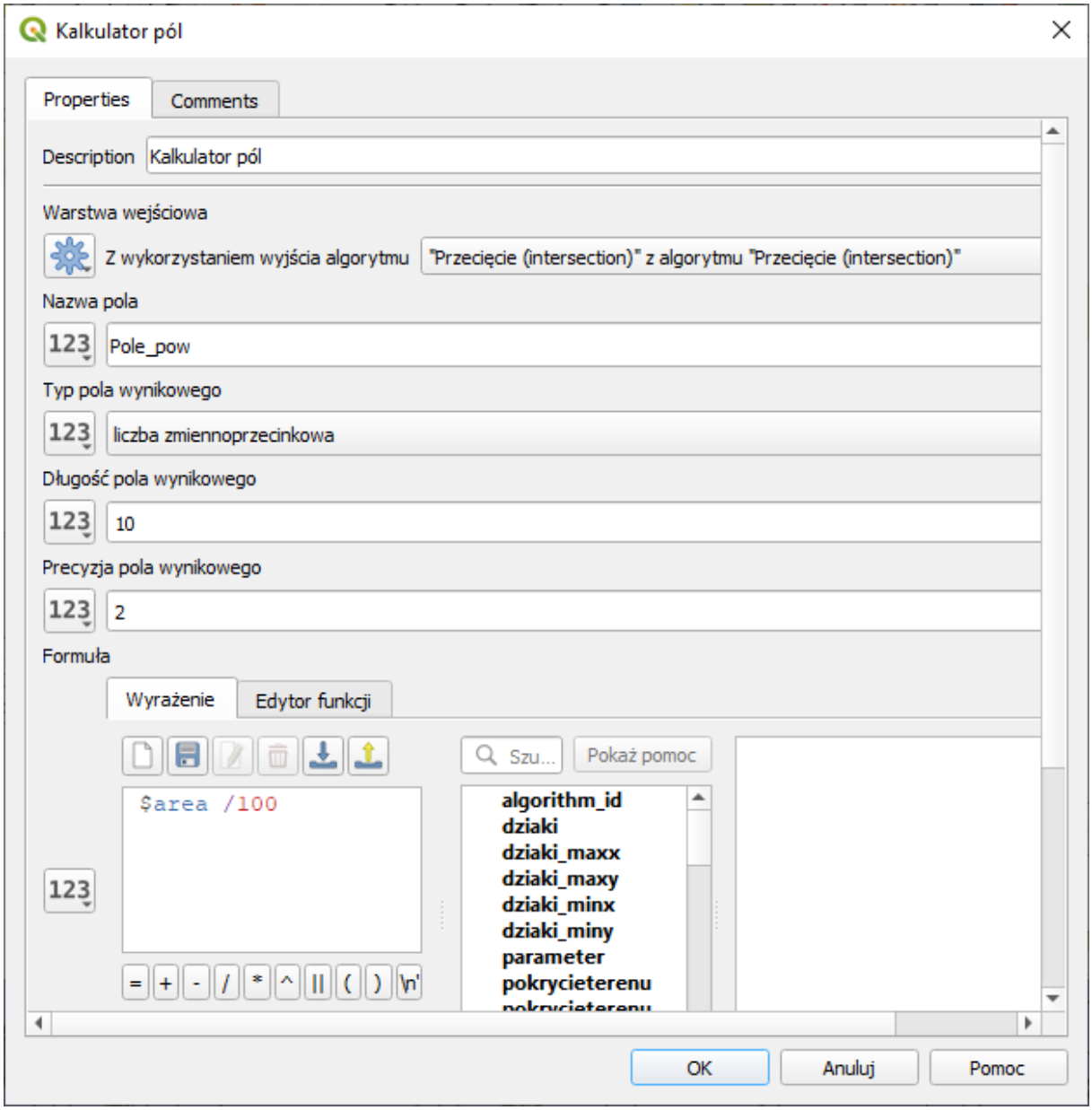
10. Kliknij **OK**. Algorytm połączy się z wprowadzoną wcześniej do modelu warstwą wektorową.
11. Dodaj kolejną warstwę wektorową i nazwij ją **Działki**:



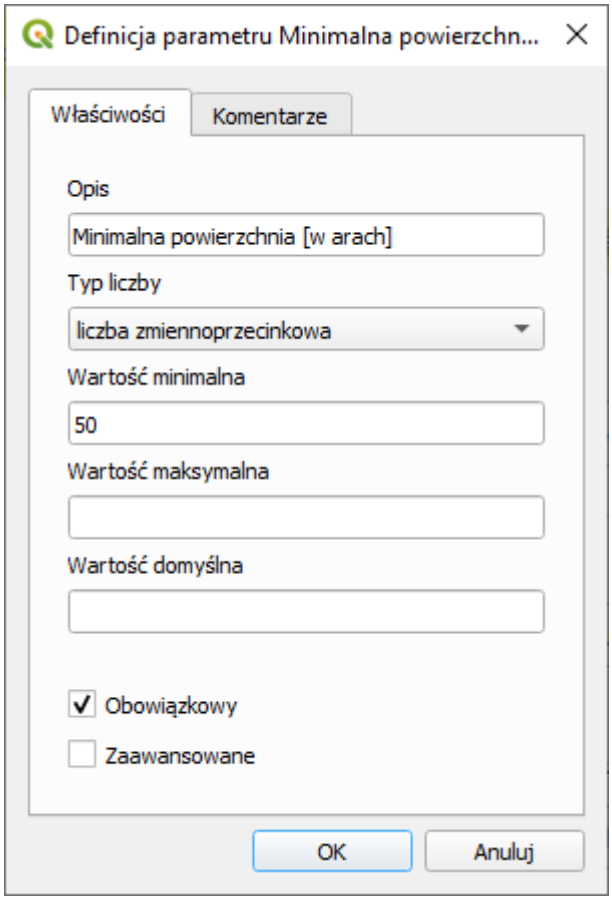
12. Odszukaj algorytm **Przecięcie (intersection)**. Ustaw:
    a. Warstwa wejściowa: Wyjście algorytmu
    b. Warstwa nakładki: Wejście modelu



13. Kliknij OK. Algorytm połączy się z **Wyodrębnij po atrybucie** oraz z blokiem **Działki**.
14. Wynikiem tego etapu będzie iloczyn warstwy działek z poligonami reprezentującymi łąki zgodnie z bazą CLC. W następnym kroku musimy policzyć w arach powierzchnię powstałych fragmentów działek spełniających pierwsze kryterium (pokrycia terenu).
15. Odszukaj narzędzie **Kalkulator pól** w sekcji **Algorytmy**.
16. W ustawieniach algorytmu podajemy nazwę i typ danych jakie zostaną obliczone i zapisane w nowej kolumnie. Nazwijmy ją **Pole_pow** (zmiennoprzecinkowa, długość 10, precyzja 2 (zaokrąglona do dwóch miejsc po przecinku).
17. Wprowadź funkcję obliczającą powierzchnię, czyli **$area** i podziel ją przez **100**, aby wynik otrzymać w arach. Okno powinno wyglądać następująco:

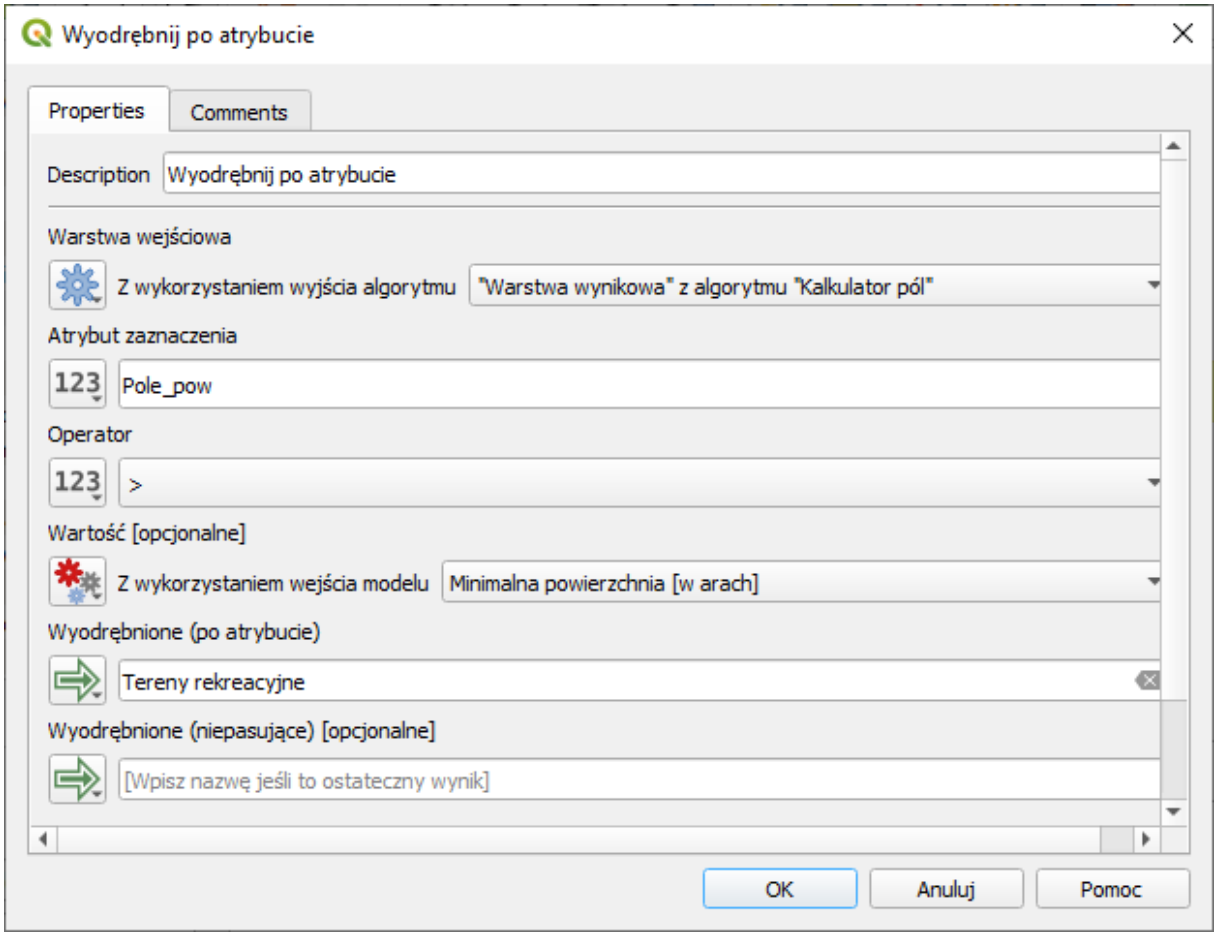


18. Kliknij **OK**. W modelu pojawi się kolejny blok – Kalkulator pól.
19. Założyliśmy, że powierzchnia będzie deklarowana przez użytkownika, zatem musimy dodać dodatkowe wejście do modelu. Odszukaj i przeciągnij **Liczba** (panel **Wejście**).
20. Uzupełnij:
    a. Opis: Minimalna powierzchnia [w arach],
    b. Typ: liczba zmiennoprzecinkowa,
    c. Wartość minimalna: 50 (zgodnie z założeniami),
    d. Zaznacz jako obowiązkowy.

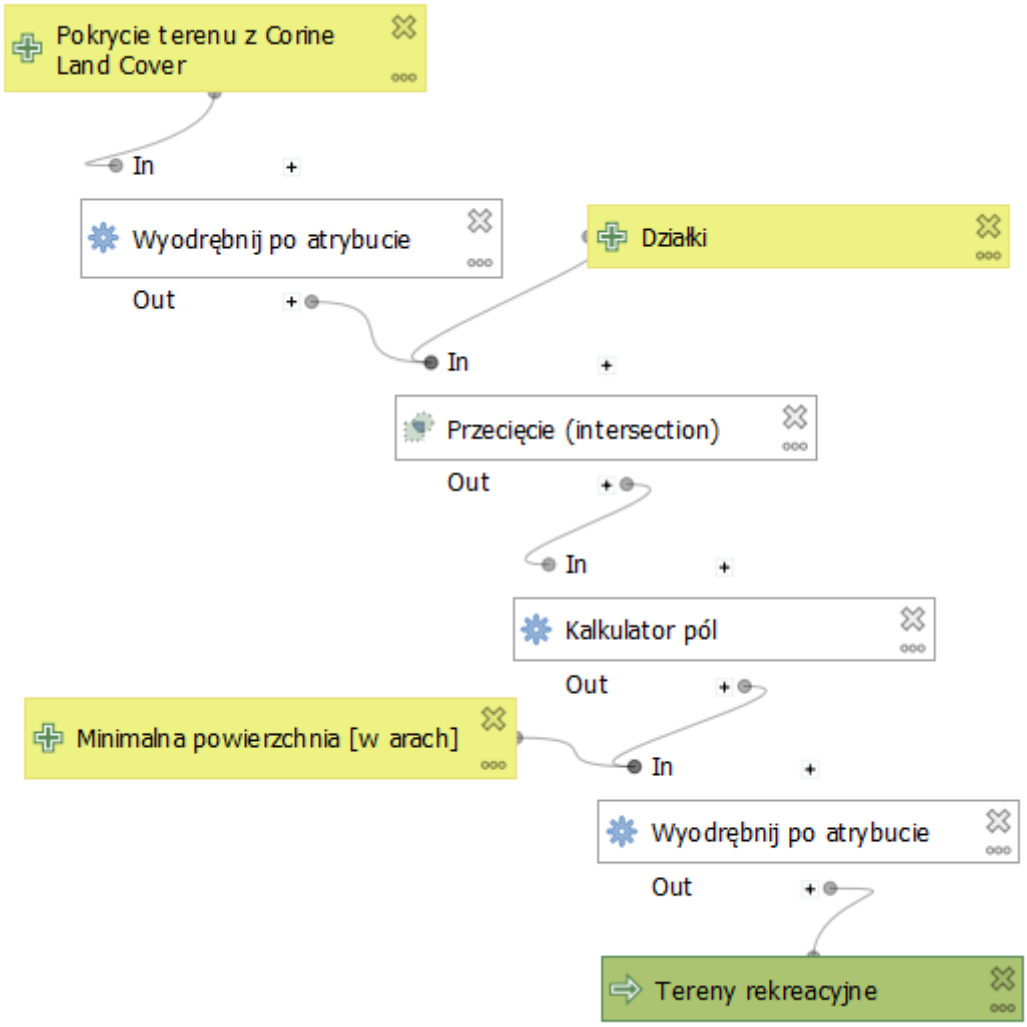


21. Dodaj do modelu kolejny raz algorytm **Wyodrębnij po atrybucie**.
22. Ustaw:
    a. Warstwa wejściowa: Wyjście algorytmu Kalkulator pól,
    b. Atrybut zaznaczania: Pole_pow (musisz być pewny, że nazwa jest identyczna z wprowadzoną przed chwilą w Kalkulatorze pól),
    c. Operator: >
    d. Wartość: Wejście modelu – Minimalna powierzchnia [w arach],
    e. Wyodrębnione: Tereny rekreacyjne

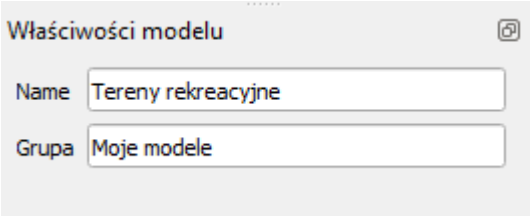
    Podajemy nazwę z jaką będzie dodawany do projektu wynik analizy.



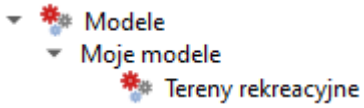
23. Utworzony model ustaw w taki sposób, aby kolejne etapy były przejrzyste.



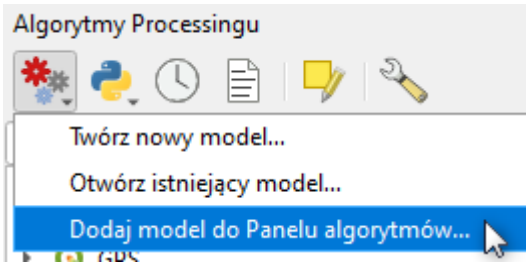
24. W ostatnim kroku należy model zapisać.
25. Przejdź do panelu właściwości modelu i uzupełnij nazwę oraz grupę, na przykład:



26. Następnie wybierz menu **Model → Zapisz model jako**…
27. Podaj nazwę np. **tereny_rekreacyjne** i zapisz plik w katalogu **ROBOCZE**.
28. Zapisany model pojawi się w panelu Algorytmy processingu:



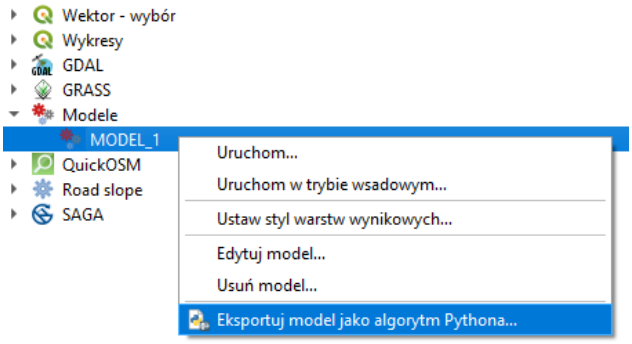
29. Jeżeli sekcja **Modele** nie jest widoczna, w pasku narzędziowym panelu **Algorytmy Processingu** rozwiń pierwsza ikonę i wybierz **Dodaj model do Panelu algorytmów**….



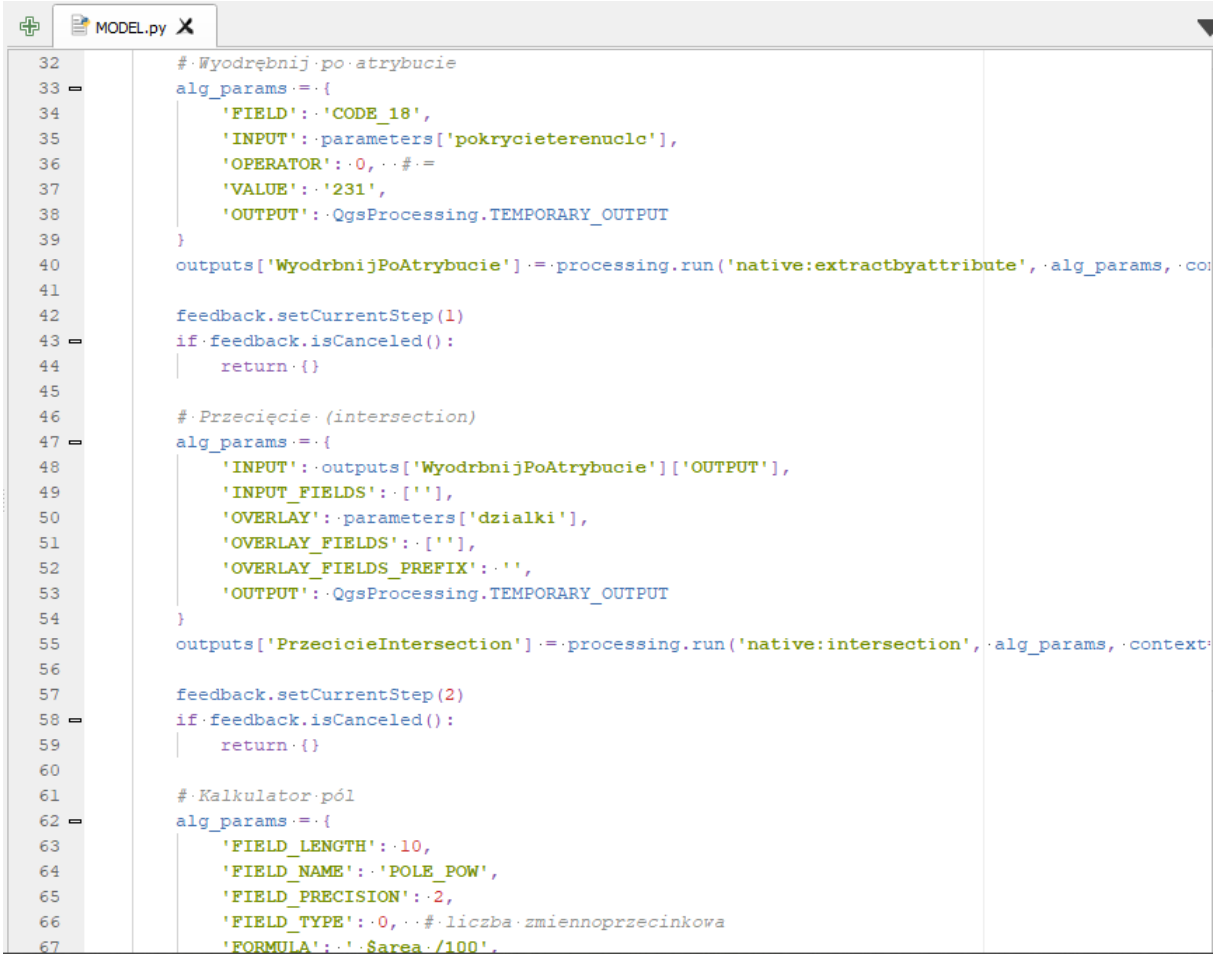
30. Podaj ścieżkę gdzie został model zapisany i wskaż zapisany przed chwilą plik: **Tereny_rekreacyjne.model3**.
31. Uruchom model podwójnym kliknięciem.
32. W oknie dialogowym stworzonego narzędzia wprowadź:
    a. Działki: DZE_Lewin_Klodzki
    b. Pokrycie terenu z Corine Land Cover: clc18
    c. Minimalna powierzchnia [w arach]: 100
33. Kliknij **Uruchom**.
34. Po wykonaniu zadania pojawi się w panelu **Warstwy** nowa warstwa wektorowa tymczasowa o nazwie **Tereny rekreacyjne**. Zawiera ona 212 obiektów.
35. Ponownie włącz model i wykonaj analizę dla warstwy DZE_Kudowa_Zdrój. Wybierz obszary o powierzchni minimalnej 200 arów.
36. Sprawdź ile obiektów znajduje się na dodanej warstwie (powinno ich być 36).

Uruchomienie modelu i jego edycja z okna konsoli PYTHON

37. Przejdź do okna algorytmów i kliknij w stworzony model PPM. Wybierz polecenie:



38. Zapisz model jako algorytm Python. Następnie otwórz konsolę Python i wczytaj algorytm.
39. Zapoznaj się z jego składnią oraz budową poszczególnych etapów przetwarzania danych.



40. Możesz wprowadzić w wybranych ustawieniach zmiany.
41. Zapisz zmodyfikowany skrypt.
42. Uruchom skrypt klikając:  .
43. Sprawdź wynik.

6. Podstawy budowy wtyczki dla aplikacji QGIS – Qt Designer

Wprowadzenie

Wtyczki służą do rozszerzenia standardowych możliwości danej aplikacji. QGIS wykorzystuje system wtyczek napisanych w językach C++ (wymagają kompilacji całego środowiska QGIS) oraz Python. QGIS wykorzystuje system repozytoriów w celu rozpowszechniania wtyczek Python. 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.

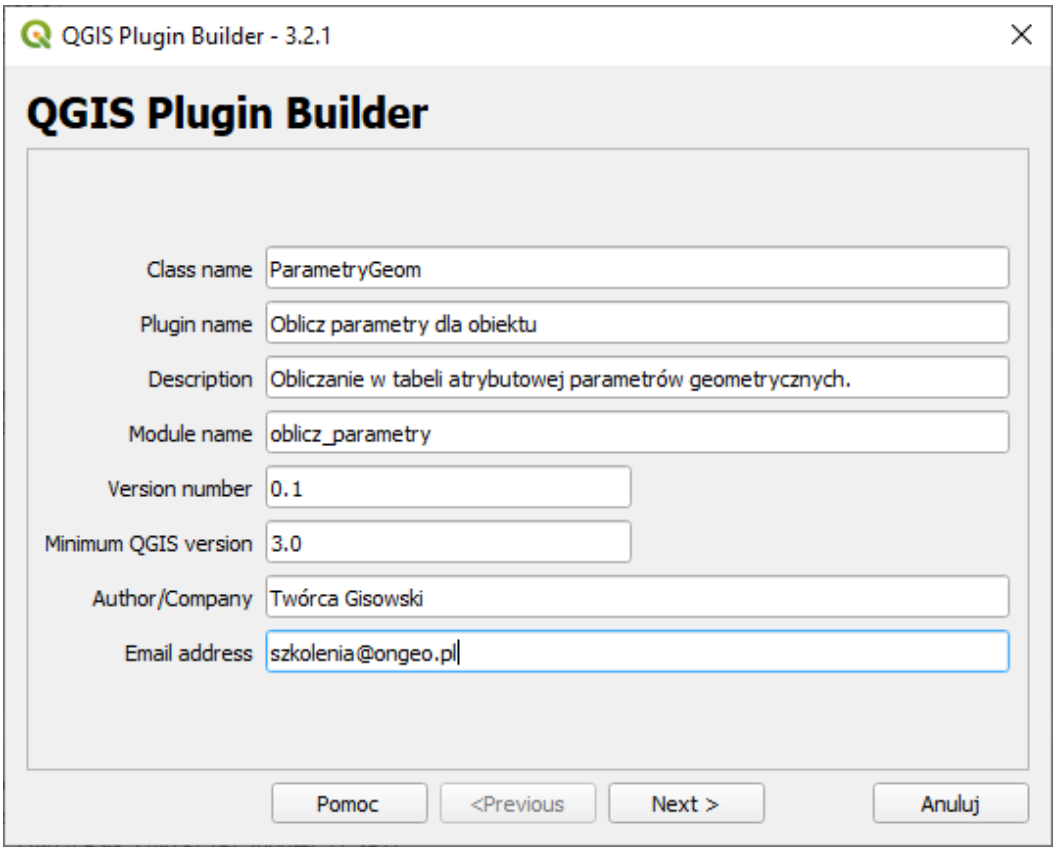
Narzędzie Plugin Builder

Narzędzie upraszczające tworzenie wtyczek poprzez wygenerowanie podstawowych plików i zasobów niezbędnych do stworzenia wtyczki. Działa w formie kreatora, gdzie przechodząc przez kolejne okna ustawiane są podstawowe informacje dotyczące metadanych i wyglądu wtyczki.

Ćwiczenie 15. Instalacja rozszerzenia QGIS Plugin Builder i konfiguracja szablonu wtyczki

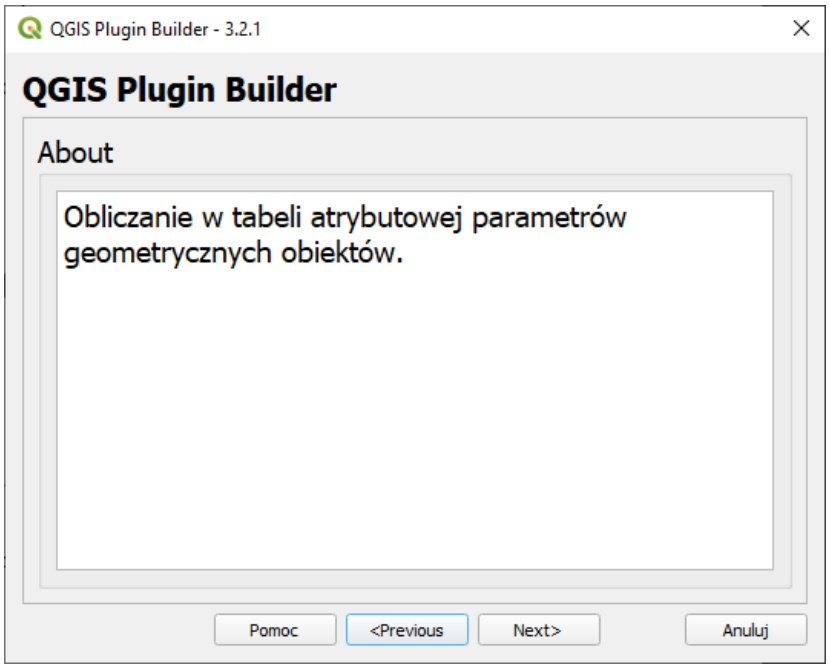
44. W oficjalnym repozytorium wtyczek odszukaj i zainstaluj Plugin Builder.
45. Sprawdź obecność ikony  na pasku narzędziowym wtyczek. Klikając w tę ikonę – uruchom narzędzie.

46. Zadaniem do wykonania w ramach opracowania wtyczki będzie wykonanie prostej funkcjonalności pozwalającej na wyliczenie parametrów geometrycznych (powierzchnia, obwód lub współrzędne) dla wskazanej warstwy wektorowej.
47. W nowo otwartym oknie uzupełnij metadane wtyczki:

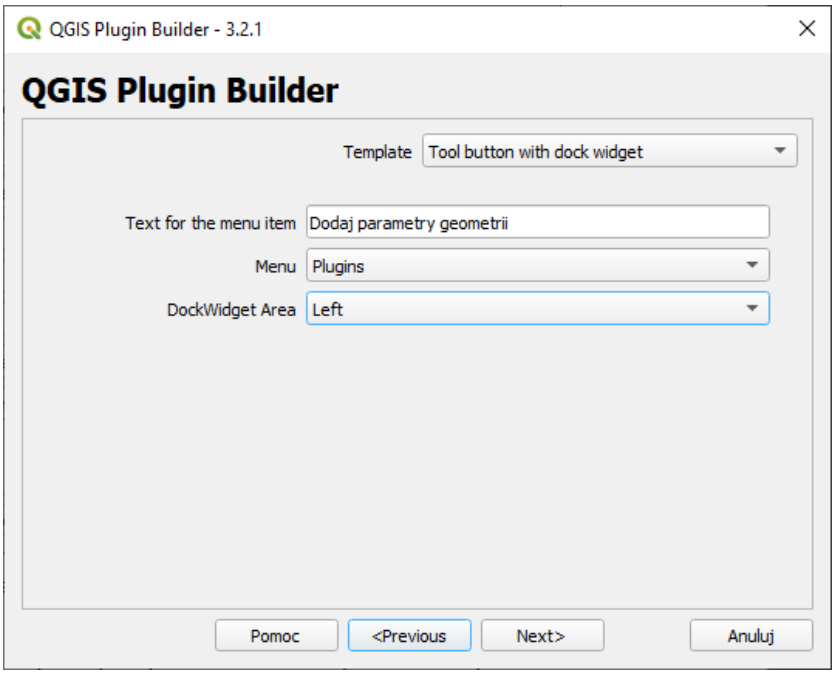


- Class name - nazwa klasy reprezentującej w QGIS daną wtyczkę. Nazwa nie może zawierać spacji oraz znaków specjalnych.
- Plugin name - Nazwa wtyczki, wyświetlana m.in. w Menedżerze wtyczek i menu.
- Description - krótki, jednolinijkowy opis wtyczki wyświetlany w Menedżerze wtyczek.
- Module name - nazwa 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.

48. Opis funkcjonalności w dłuższej formie można podać w kolejnym oknie w polu About:



49. W kolejnym kroku można wybrać rodzaj okna dialogowego:



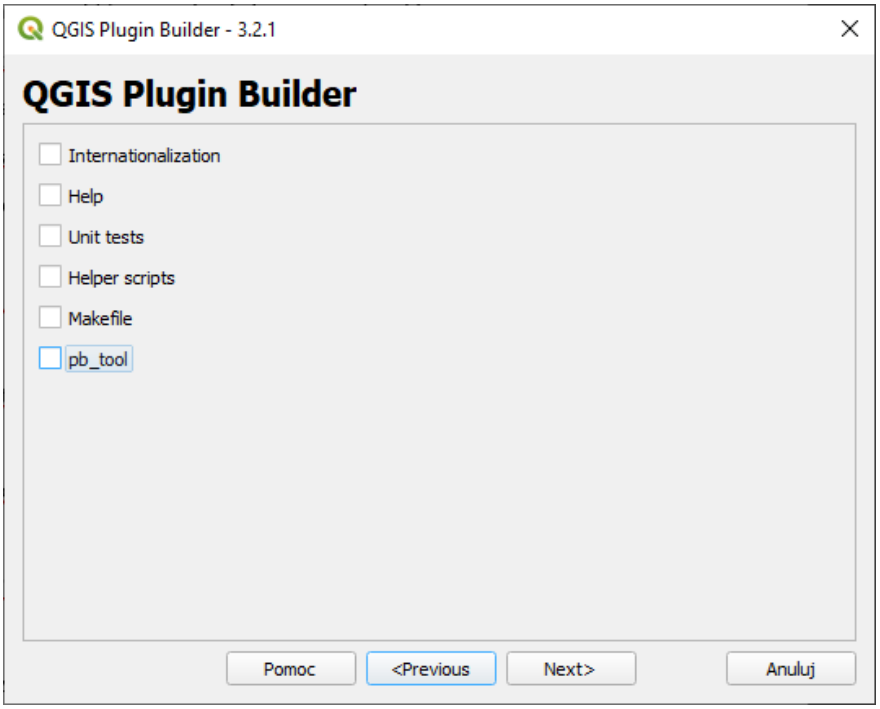
- Tool button with dialog - przycisk na pasku narzędzi, który wyświetla osobne okno dialogowe wtyczki.

- 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.

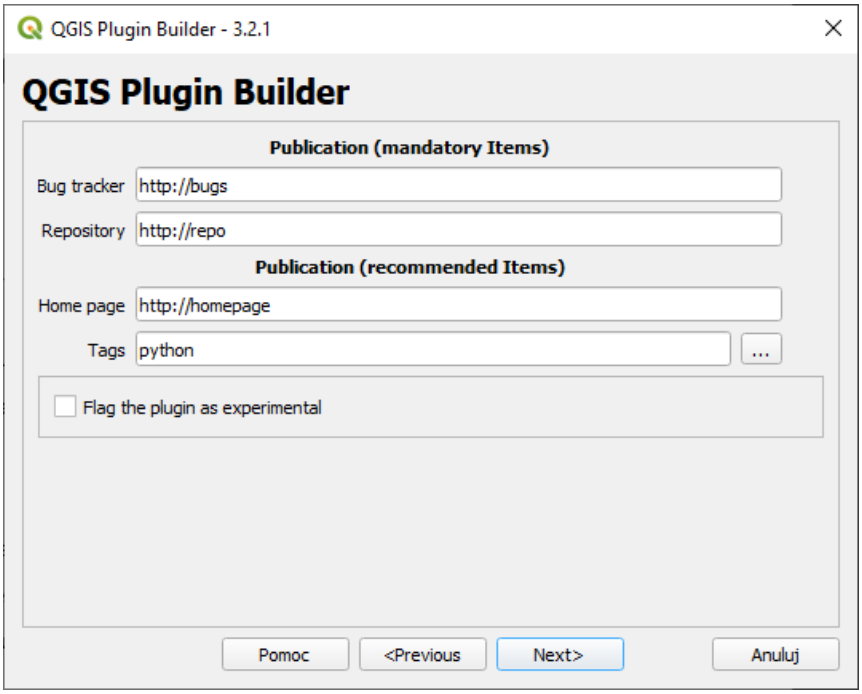
50. Dodatkowych elementów nie musimy dodawać, jednak można je wybrać w tzw.: ‘czekboksach”.



- Internationalization - dodaje katalog i pliku 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.

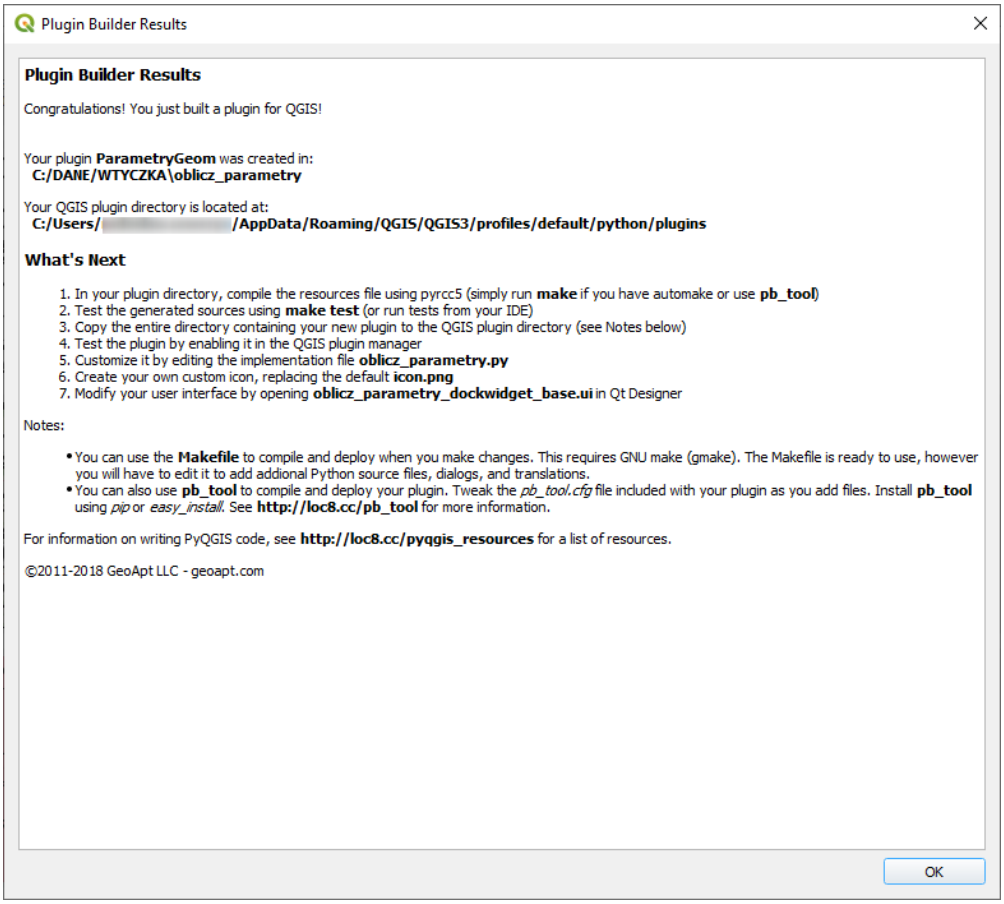
51. Dodatkowe informacje są wymagane, jeśli wtyczka ma zostać opublikowana w oficjalnym repozytorium QGIS.



- 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.

52. Kolejnym krokiem jest wskazanie katalogu wyjściowego, w którym zlokalizowana zostanie wtyczka. Wskaż w okienku ścieżkę: **….DANE\WTYCZKA.**

Po kliknięciu przycisku Generate otrzymamy podsumowanie wraz z najważniejszymi informacjami:



Struktura wtyczki

- **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**

Plik XML programu Qt Designer określający ścieżki do zasobów wtyczki np. ikon. Po każdej

modyfikacji tego pliku należy go skompilować do pliku .py za pomocą polecenia pyrcc5 w 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.

Aby wskazać zasób należy podać prefiks wraz z nazwą zasobu poprzedzone dwukropkiem:

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

- **Plik .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 .

- **__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.

- **Plik .py**

Plik .py, który jest importowany w __init__.py 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. 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 16. Przenoszenie i kompilacja wtyczki

Skopiuj katalog utworzonej wtyczki do folderu, w którym QGIS przechowuje zainstalowane wtyczki. Przed jej pierwszym uruchomieniem skompiluj zasoby z pliku resource.qrc .

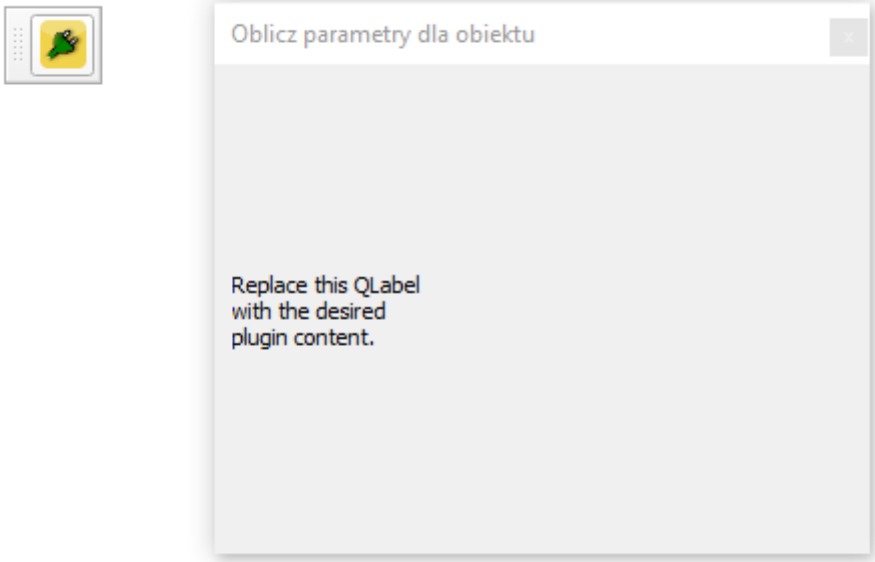
53. Katalog wtyczek można znaleźć wybierając w menu **QGIS Ustawienia -> Profile użytkownika -> Otwórz katalog aktywnego profilu** i przechodząc do katalogu **python/plugins**. Jeśli katalog **python** nie istnieje to można go utworzyć ręcznie. Następnie kopiujemy wtyczkę do tego folderu.
54. W celu kompilacji wtyczki należy uruchomić konsolę OSGeo4W Shell i wywołać po kolei dwa polecenia:
    a. **py3_env** - ustawienie Python3 jako aktywnego (jeśli to konieczne w danej wersji QGIS/Python)
    b. **qt5_env** - ustawienie narzędzie Qt 5 jako aktywnych

    Następnie wpisujemy polecenie:

```
pyrcc5 -o C:/.../resources.py C:/.../resources.qrc
```

Ważne: Należy podać pełne ścieżki do plików, można sobie ułatwić zadanie poprzez przeciąganie plików na okno konsoli, dzięki czemu pełne ścieżki zostaną wpisane w miejscu kursora.

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

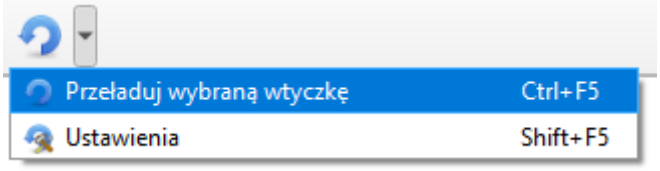


Plugin Reloader

Narzędzie umożliwia przeładowanie wskazanej wtyczki w QGIS bez konieczności resetowania QGIS lub ręcznego jej wyłączania/włączania w Menedżerze wtyczek .

Ćwiczenie 17. Instalacja i konfiguracja Plugin Reloader

56. Poprzez okno zarządzania wtyczkami zainstaluj i skonfiguruj **Plugin Reloader** aby możliwe było resetowanie wtyczki dodanej z poziomu wiersza poleceń.
57. Sprawdź czy narzędzie działa:



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 .

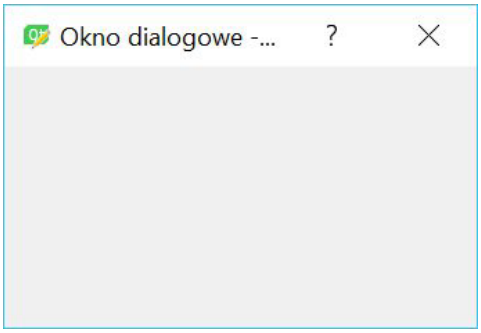
Import z głównych modułów:

```
# 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 *
```

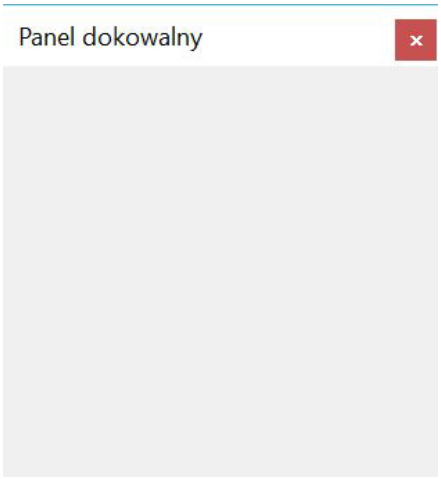
Widżety

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.



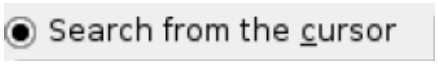
- QPushButton – przycisk

- QLabel – etykieta tekstowa

- QLineEdit – okienko do wpisywania tekstu (jednoliniowe)

- QTextEdit – okienko do wpisywania tekstu (wieloliniowe)

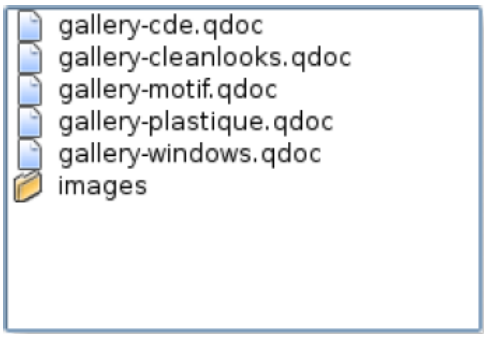
- QRadioButton – przycisk opcji



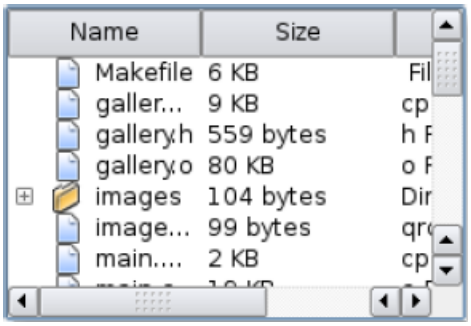
- QCheckBox – przycisk wyboru

- QComboBox – lista wyboru

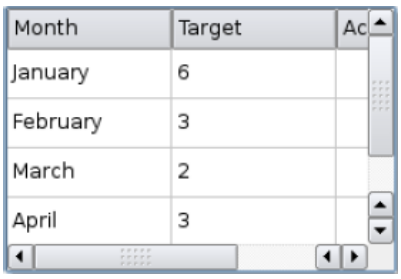
- QListWidget – lista elementów
- QListWidgetItem – element listy



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



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



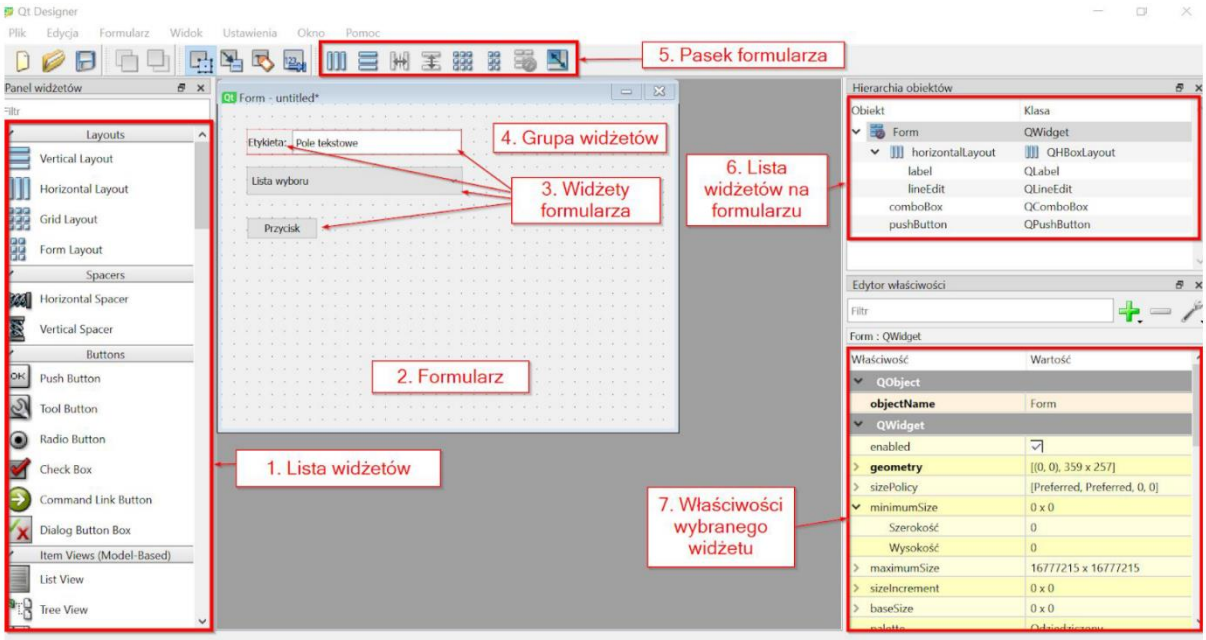
Qt Designer umożliwia również rejestrowanie dodatkowych widżetów. W ten sposób można również dodać wiele kontrolek pochodzących z QGIS.

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. Aplikacja jest instalowana razem z QGIS i jest dostępna w menu Start.

Okna dialogowe stworzone w programie Qt Designer zapisywane są w formacie UI (XML).

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
- 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 - układa widżety w dwóch kolumnach, lewa zawiera etykiety, prawa kontrolki edycyjne,
- Usuń rozmieszczenie - pozwala usunąć wskazany układ z formularza. do zmiany szerokości,

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

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:

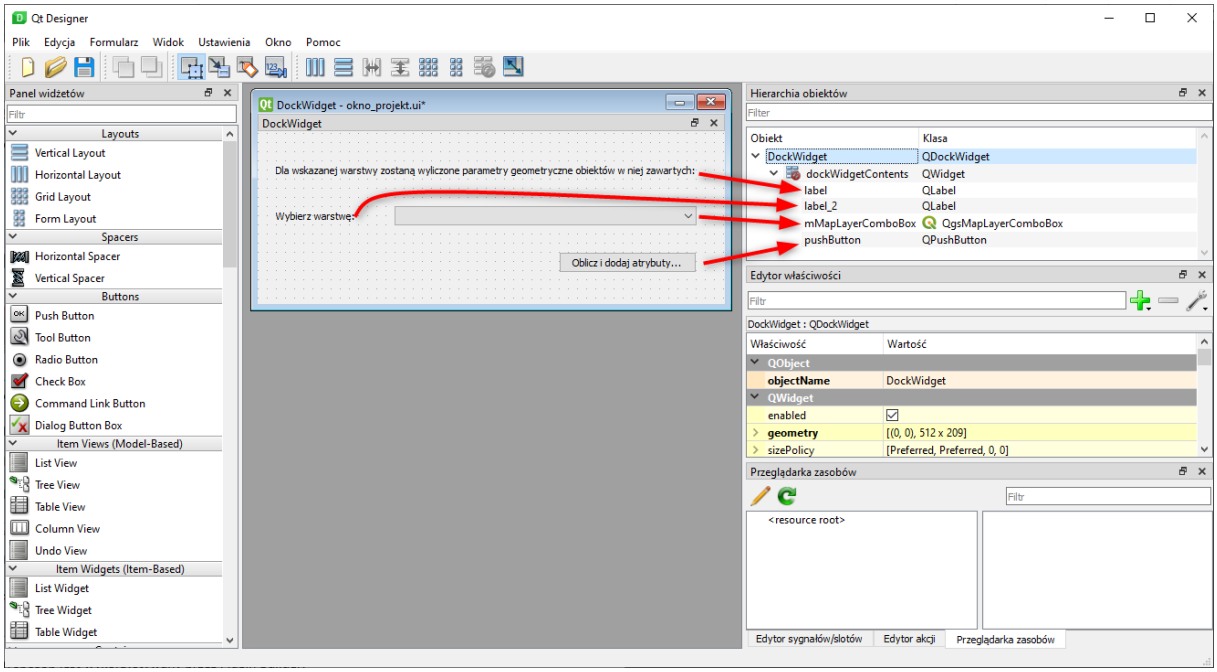
```
# uic służy do dynamicznej kompilacji plików .ui do klas Pythona

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.

Ćwiczenie 18. Tworzenie nowego okna wtyczki

58. Za pomocą narzędzia QT Designer otwórz plik **oblicz_parametry_dockwidget_base.ui** i skonfiguruj okno wg poniższego wzoru:



Do określenia układu można użyć opcji **Rozmieść w formularzu** .

59. **WAŻNE:** Zapisz plik po skonfigurowaniu okna!!!

Na formularz należy dodać 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.

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)
```

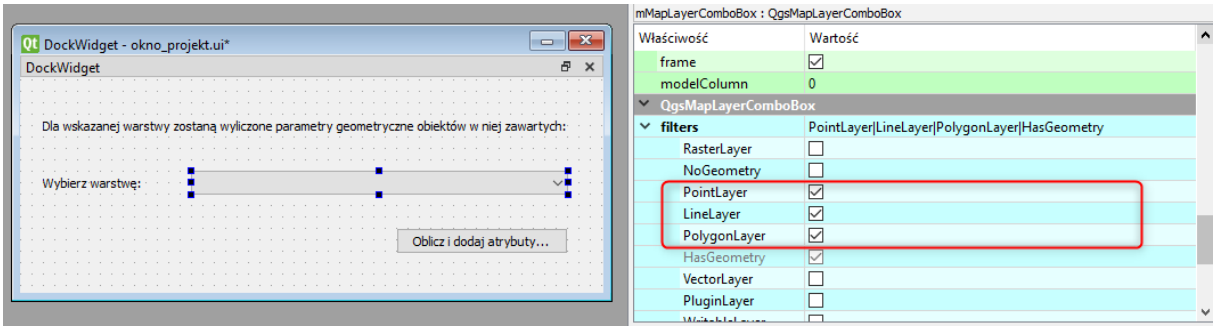
QgsMapLayerComboBox

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

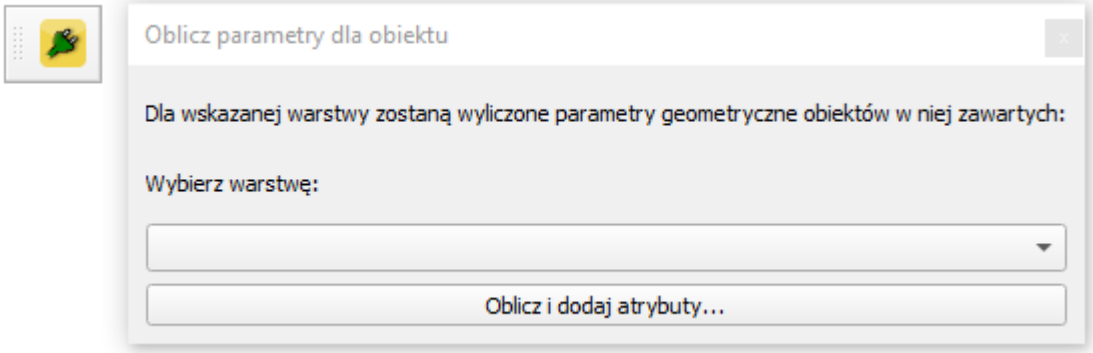
- setLayer(QgsMapLayer), currentLayer() - ustawia i zwraca wybraną warstwę.
- setFilters(QgsMapLayerProxyModel.Filters), filters() - ustawia i zwraca filtr warstw.
- setShowCrs(bool), showCrs() - ustawia i zwraca informację, czy przy warstwach ma się wyświetlać ich układ współrzędnych.

Ćwiczenie 19. Ograniczenie typu warstw wyświetlanych na liście wyboru

60. Zaznacz w oknie obiekt reprezentujący listę wyboru QgsMapLayerComboBox.
61. Przejdź do jego właściwości i odszukaj opcje umożliwiające ustawienie filtrowania danych ograniczające wynik do warstw wektorowych:



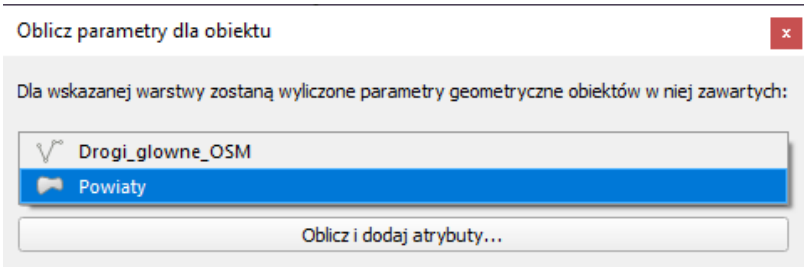
62. Sprawdź, czy po przeładowaniu wtyczki okno dalej działa. Jeśli nie – przywróć ustawienia takie jakie były domyślnie i przeładuj wtyczkę.
63. Po wykonaniu zmian – okno wtyczki w aplikacji QGIS wygląda tak:



64. Aby wtyczka została podłączona do funkcji QGIS i uzyskała dostęp m.in. do warstw z tabeli zawartości – należy zmodyfikować plik oblicz_parametry.py, który znajduje się w folderze wtyczki:
C:\Users\(konto)\AppData\Roaming\QGIS\QGIS3\profiles\default\python\plugins\oblicz_parametry
Dodaj następujące elementy do składni pliku Pythona:

```
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/
"""
from qgis.PyQt.QtCore import QSettings, QTranslator, QCoreApplication, Qt
from qgis.PyQt.QtGui import QIcon
from qgis.PyQt.QtWidgets import QAction
from qgis.core import QgsProject

# Initialize Qt resources from file resources.py
from .resources import *

# Import the code for the DockWidget
from .oblicz_parametry_dockwidget import ParametryGeomDockWidget
import os.path
```

65. Przeładuj wtyczkę i dodaj do widoku mapy dwie dowolne warstwy shp – czy menu wyboru uzupełniło się w dwie nowe pozycje?



66. Wróć do okna QT Designer. Zapoznaj się z edycją sygnałów i slotów w oknie wtyczki. Sprawdź możliwości przekazywania informacji pomiędzy poszczególnymi pozycjami.
67. Wykonaj – wg instrukcji przekazywanych przez prowadzącego próbę skonfigurowania działania wtyczki tak, aby na podstawie przygotowanego wcześniej skryptu **podstawowe_atr_geom.py** możliwe było wykonanie obliczeń za pośrednictwem stworzonej wtyczki.

    Edycji mogę być poddane pliki:

    __init__.py

    oblicz_parametry.py

    oblicz_parametry_dockwidget.py

    oblicz_parametry_dockwidget_base.ui
68. Odświeżając działanie wtyczki – sprawdź jej funkcjonalność.
69. Możesz zmodyfikować wygląd jej okna dodając własne elementy i funkcje.