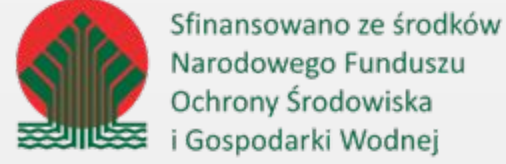
Programowanie Python w QGIS
Poziom podstawowy

Piotr Pociask
www.gis-support.pl



Agenda

1. Omówienie budowy aplikacji QGIS
2. Interfejs programowania aplikacji QGIS
3. Obsługa danych rastrowych
4. Obsługa danych wektorowych
5. Omówienie Narzędzi geoprocessingu
6. Skrypty Narzędzi geoprocessingu
7. Podstawowe informacje o budowie i działaniu wtyczek QGIS
8. Biblioteka Qt i tworzenie interfejsów graficznych

Open Source GIS

Platforma QGIS

QGIS Desktop

QGIS Server

QField (Android)

QGIS Web Client

Lizmap

QGIS ma budowę modułową. Biblioteki zawierają API, z których korzysta aplikacja do wykonywania operacji.

Aplikacja QGIS np. qgis.exe

Biblioteki QGIS:

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

Zewnętrzne biblioteki:

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

QGIS jest napisany w języku programowania C++. Dzięki tzw. *bindings* możliwe jest wykorzystanie ich możliwości za pomocą języka Python.

Biblioteki QGIS (C++):

- qgis.core
- qgis.gui
- qgis.analysis
- ...

Zewnętrzne biblioteki:

- Qt
- GDAL/OGR
- proj4
- GEOS
- ...

Python bindings

generowane automatycznie z kodu C++

Kod użytkownika:

- konsola
- wtyczki
- skrypty
- funkcje
- …

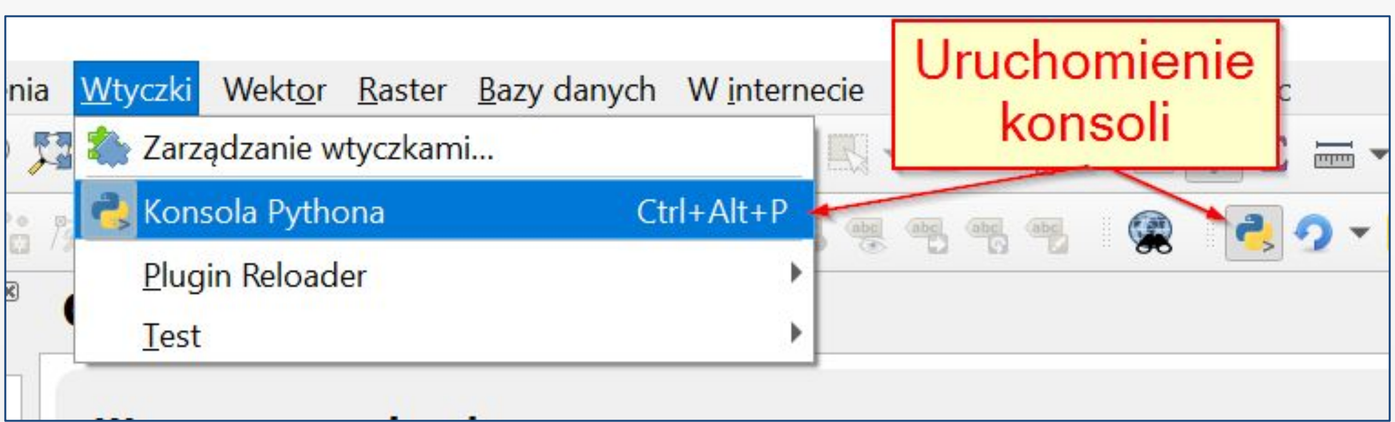
from qgis.**core** import ...

Aplikacja QGIS wykorzystuje Pythona jako język skryptowy. 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:

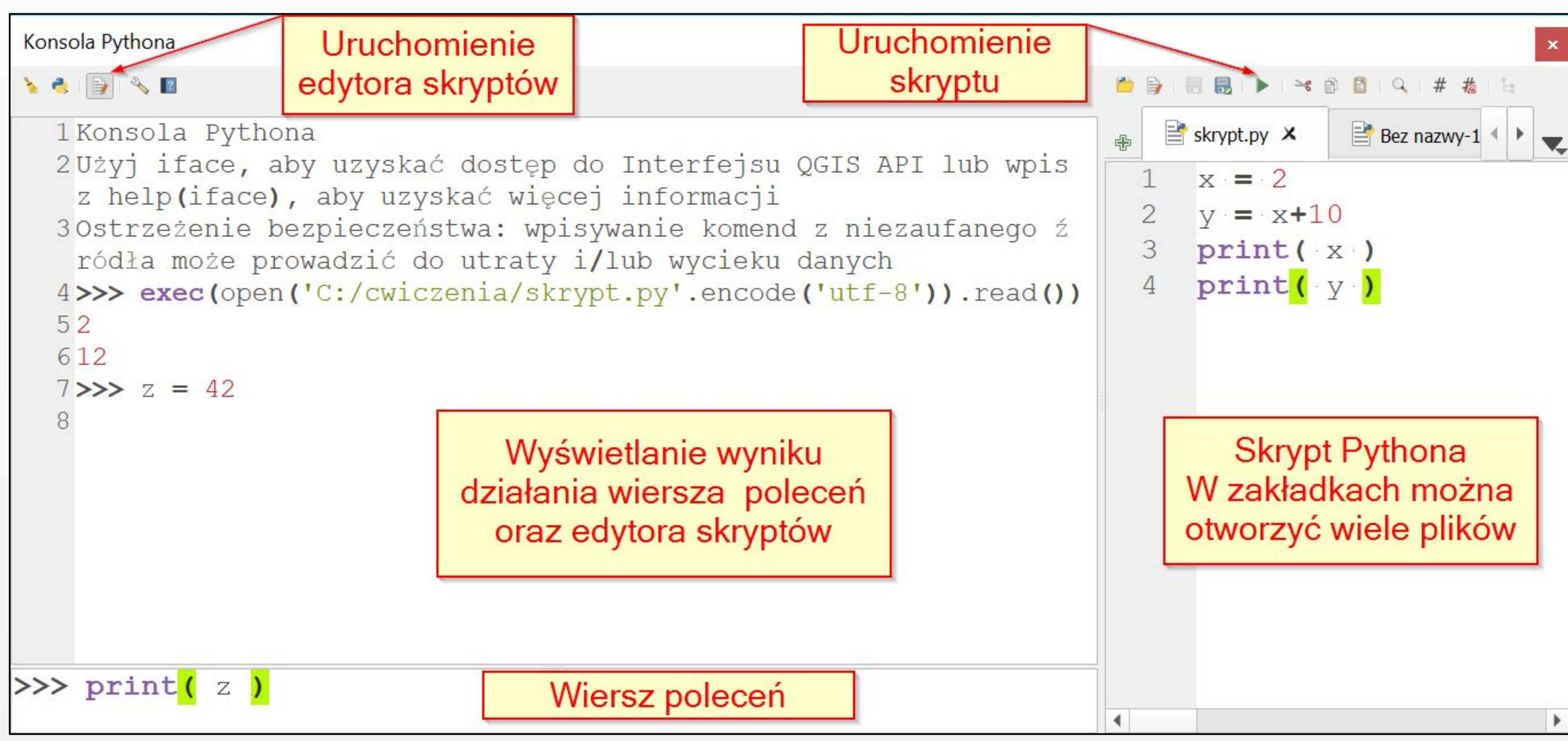
- wiersz poleceń umożliwiający wykonywanie poleceń po wpisaniu ich w konsoli,
- *Edytor skryptów* umożliwiający pisanie i uruchamianie plików z kodem źródłowym.

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.

Aby uruchomić konsolę należy w QGIS z menu *Wtyczki* wybrać polecenie *Konsola Pythona* lub kliknąć odpowiedni przycisk na pasku narzędzi.



Konsola Pythona w QGIS



Interfejs programowania aplikacji QGIS

Dokumentacja QGIS API

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

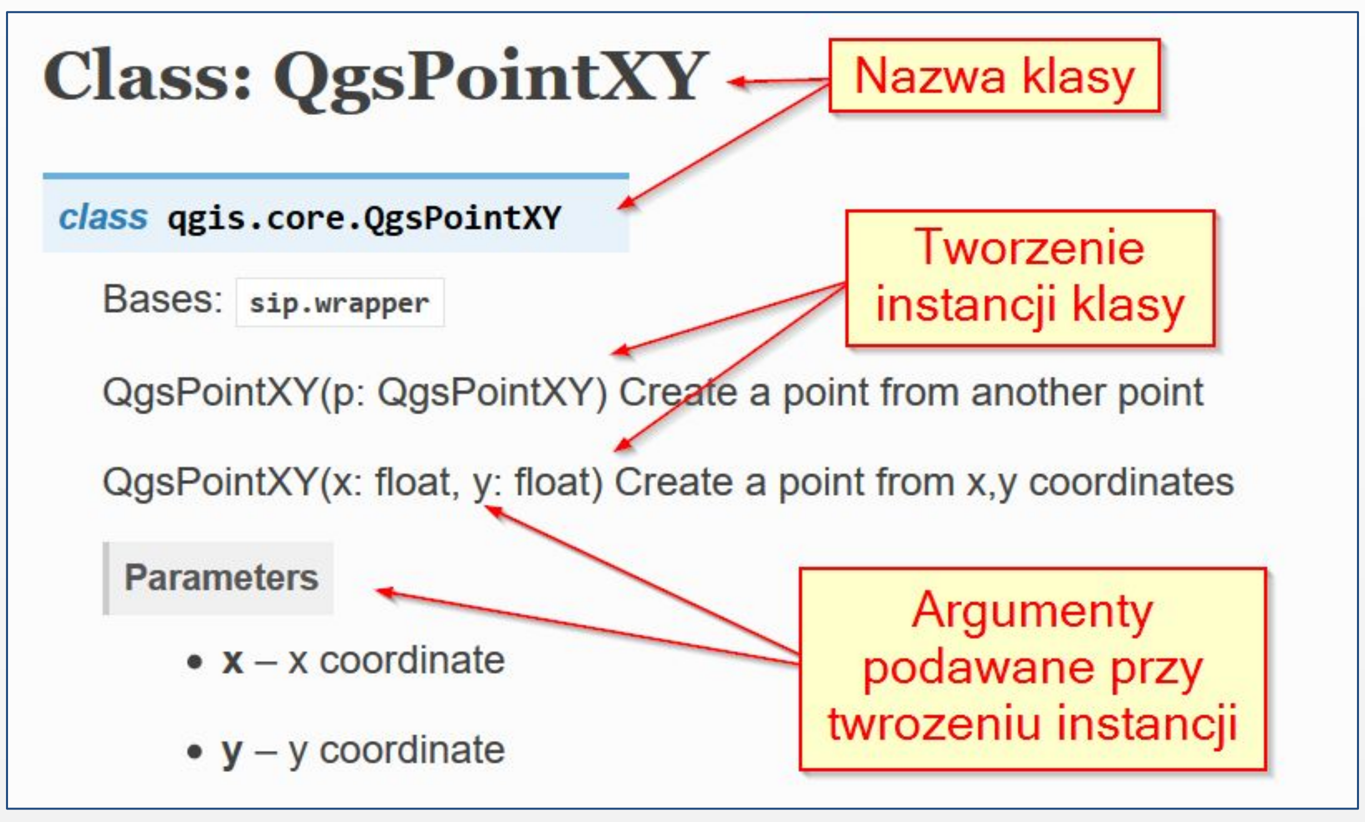
http://www.qgis.org/pyqgis - dokumentacja dla Pythona

Na obu stronach możliwe jest wybranie konkretnej wersji QGIS: C++ od 1.6, Python od 3.0

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

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.

QgsPointXY - klasa reprezentująca punkt w przestrzeni 2D




```
>>> punkt = QgsPointXY( 10, 15 )
```

```
>>> print( punkt )
<QgsPointXY: POINT(10 15)>
```

Jeśli wynik wpisanego polecenia znajduje się pomiędzy znakami <> oznacza to, że zwrócony został obiekt Pythona danej klasy. W zależności od rodzaju klasy informacje mogą się różnić np. dla punktu są to informacje o jego współrzędnych.

QGIS API - metody

setX(*self, x: float*) — Nazwa metody, Argumenty

Sets the x value of the point

Parameters — Opis argumentów

x – x coordinate

x(*self*) → **float** — Wynik działania

Gets the x value of the point — Opis metody

Returns — Opis zwracanej wartości

x coordinate

```
>>> punkt.setX( 20.5 )
>>> print( punkt.x() )
20.5
```

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.

Główne 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 qgis.core).

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)

qgis.core

- **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

- **QgsCoordinateReferenceSystem** – informacje o układzie współrzędnych
- **QgsCoordinateTransform** – pomocnicza klasa do transformacji współrzędnych pomiędzy różnymi układami odniesienia
- **QgsProject** – informacje o aktualnym projekcie, umożliwia odczytywanie/zapisywanie informacji z/do projektu oraz zarządzanie warstwami w QGIS: wczytywanie, usuwanie, listowanie, wyszukiwanie.

qgis.gui

- **QgisInterface** – specjalna klasa umożliwiająca komunikację pomiędzy Pythonem, a uruchomionym środowiskiem QGIS
- **QgsMapCanvas** – okno mapy
- **QgsMessageBar** - pozwala wyświetlać informacje użytkownikowi nad oknem mapy

W konsoli obiekt *iface* jest instancją klasy **QgisInterface** i pozwala na interakcję z aplikacją QGIS.

```
>>> print( iface )
<qgis._gui.QgisInterface object at 0x… >
```

- **activeLayer() - zwraca aktywną warstwę**

```
>>> warstwa = iface.activeLayer()
>>> print( warstwa )
<qgis._core.QgsVectorLayer object at 0x… >
```

- **mapCanvas() - zwraca okno mapy QGIS**

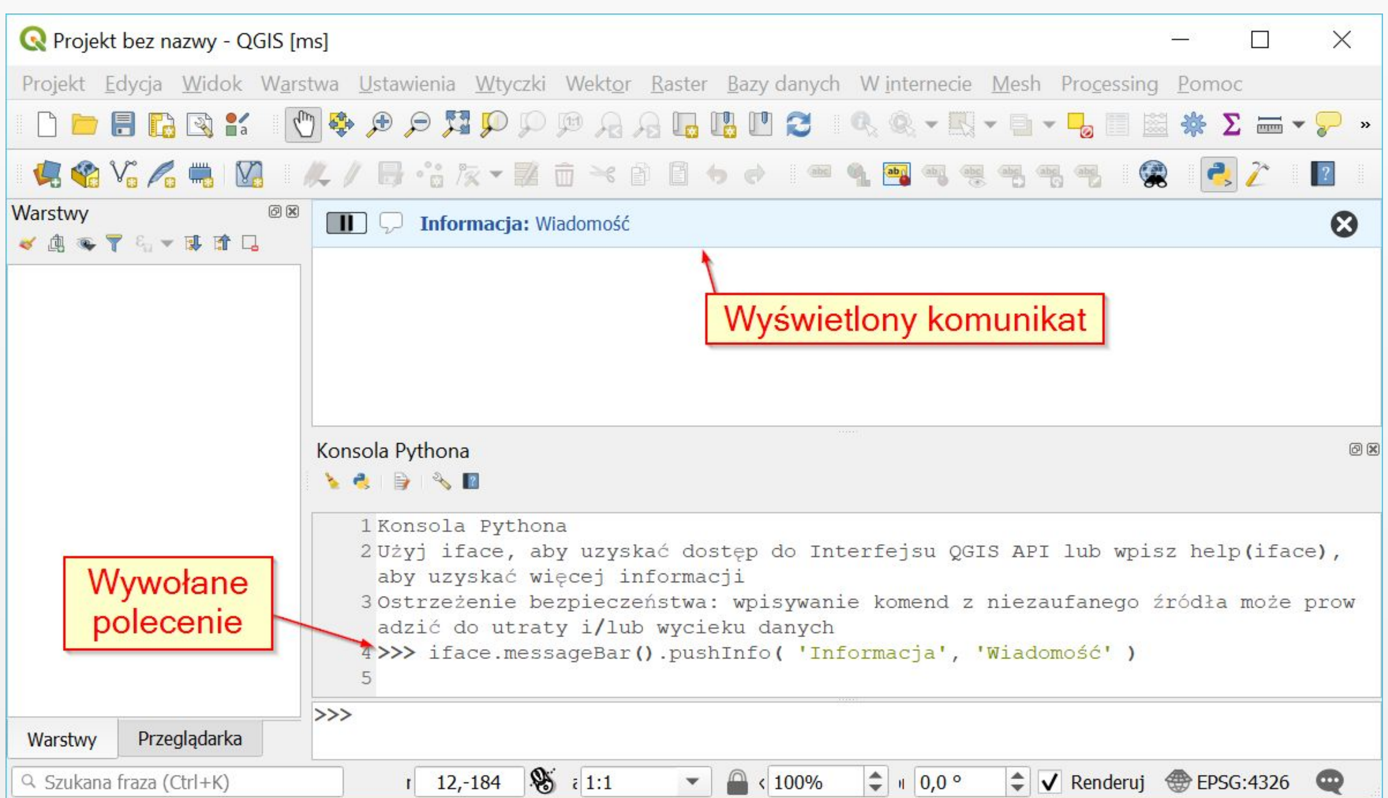
```
>>> mapa = iface.mapCanvas()
>>> print( mapa )
<qgis._gui.QgsMapCanvas object at 0x… >
```

- **messageBar() - umożliwia wyświetlanie komunikatów użytkownikowi nad oknem mapy**

```
>>> print( iface.messageBar() )
<qgis._gui.QgsMessageBar object at 0x… >
```

Dostępne są różne typy komunikatów (informacja, ostrzeżenie, błąd), które wpływają na kolor wyświetlanych okien. Aby wyświetlić informację w oknie głównym QGIS należy wykorzystać obiekt *iface.messageBar()*.

```
iface.messageBar().pushInfo( 'Informacja', 'Wiadomość' )
```



QgsMessageBar

```
iface.messageBar().pushSuccess( 'Sukces',
                'Analiza zakończona sukcesem' )
```

Sukces: Analiza zakończona sukcesem

```
iface.messageBar().pushWarning( 'Ostrzeżenie',
                'Ta operacja może być niebezpieczna' )
```

Ostrzeżenie: Ta operacja może być niebezpieczna

```
iface.messageBar().pushCritical( 'Błąd','Wystąpił błąd' )
```

Błąd: Wystąpił błąd

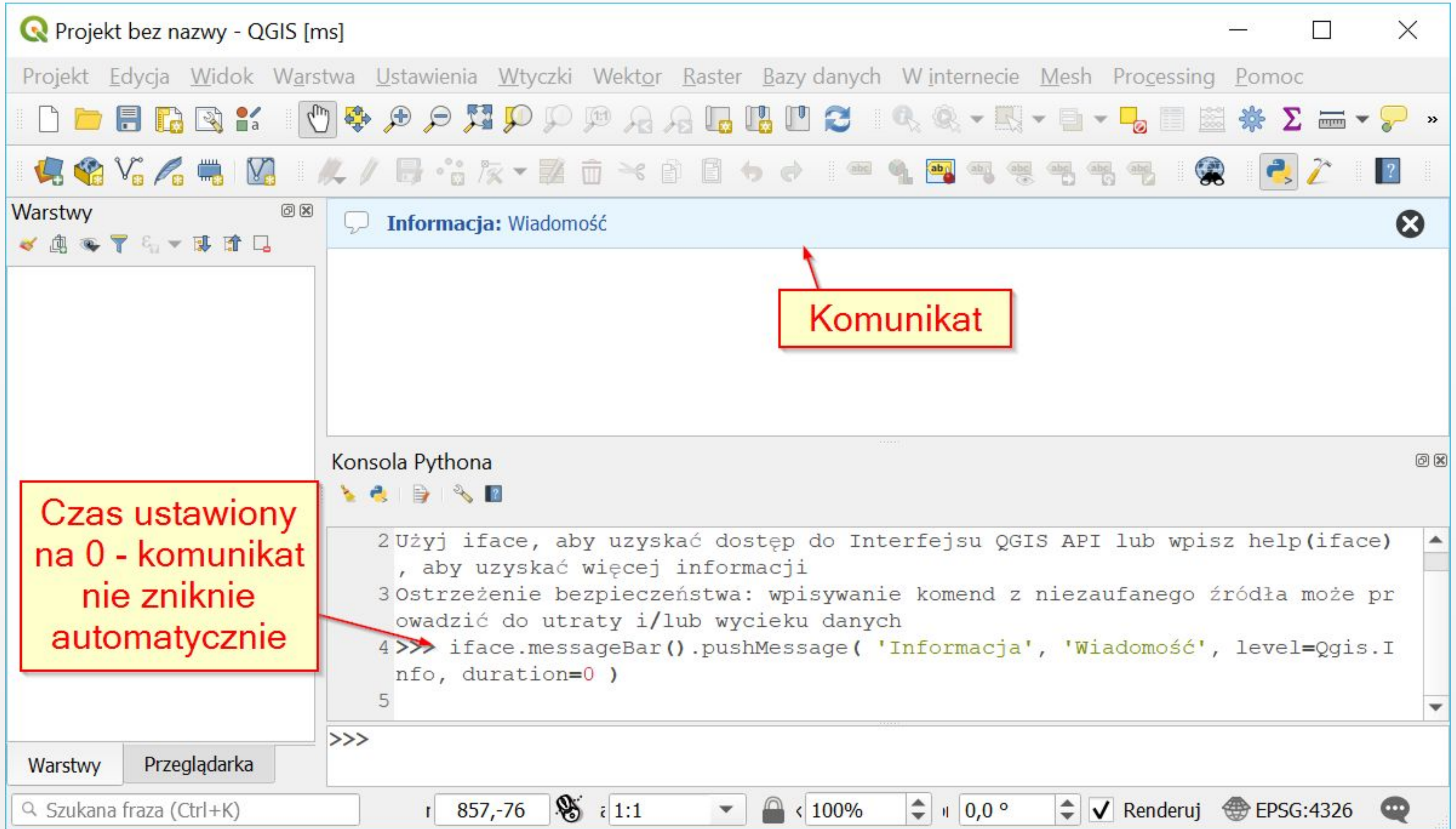
QgsMessageBar

Metoda *pushMessage()* pozwala bezpośrednio sterować typem wiadomości (argument *level*) oraz czasem jej wyświetlania (argument *duration*). Długość wyświetlania podajemy w sekundach, podanie wartości 0 powoduje, że komunikat nie zniknie automatycznie. Jako typ wiadomości podajemy jedną z wartości:

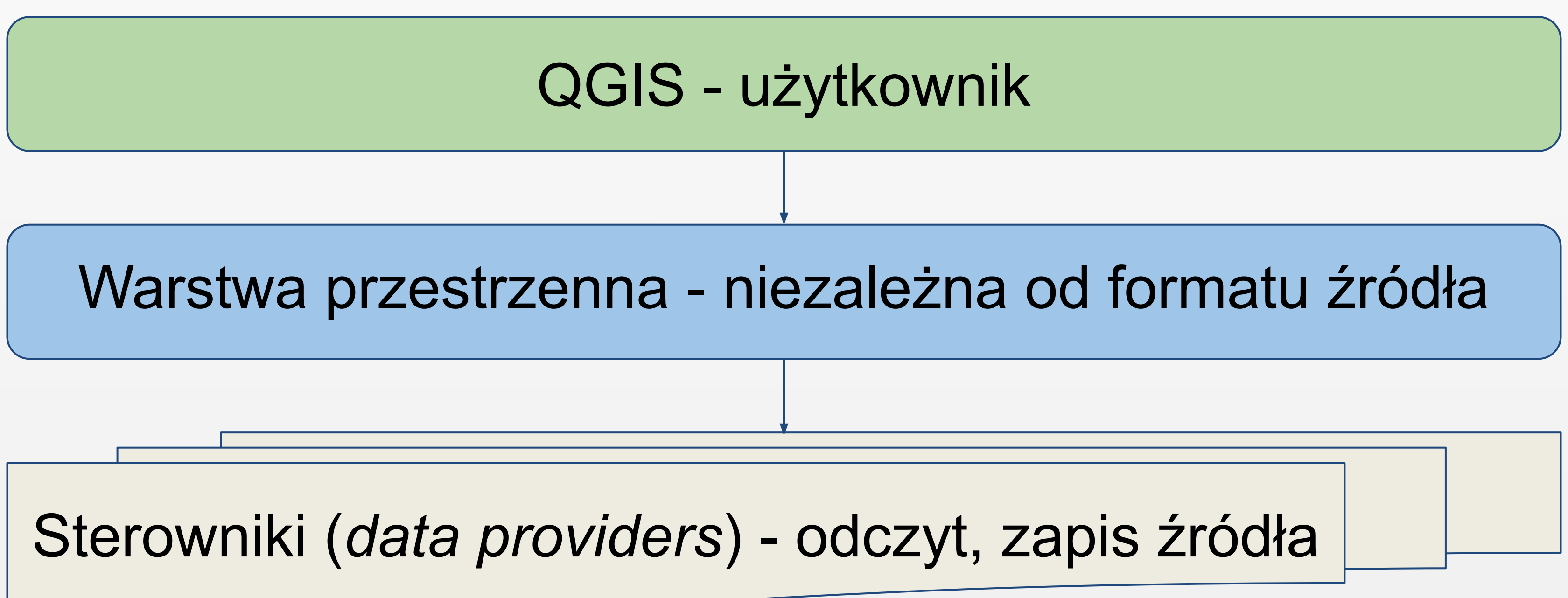
- **Qgis.Info** - niebieska informacja
- **Qgis.Success** - zielony sukces
- **Qgis.Warning** - żółte ostrzeżenie
- **Qgis.Critical** - czerwony błąd

QgsMessageBar

```
iface.messageBar().pushMessage( 'Informacja',
   'Wiadomość', level=Qgis.Info, duration=0 )
```



Sterowniki danych (ang. *data providers*) to biblioteki służące aplikacji QGIS do komunikacji ze źródłem danych m.in. odczyt, edycja, tworzenie. Źródłem danych są m.in. pliki (SHP, GML, CSV, ...), bazy danych (PostGIS, SpatiaLite, ...) lub usługi (WMS, WFS, ...).



QgsMapLayer

Klasa bazowa dla wszystkich typów warstw

QgsVectorLayer

Dane wektorowe

QgsRasterLayer

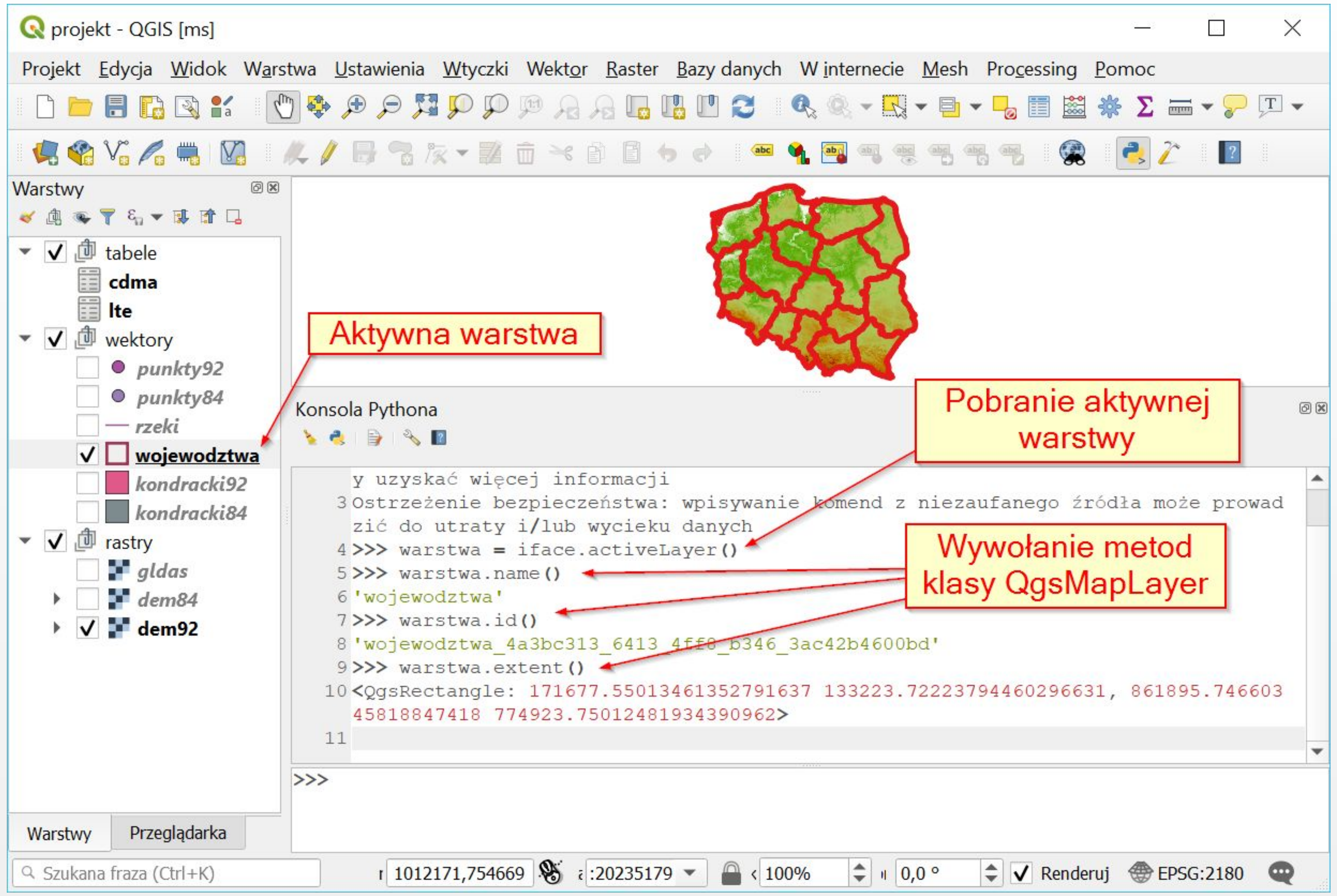
Dane rastrowe

QgsMapLayer

Metody wspólne dla wszystkich typów warstw:

- **name()** - wyświetlana nazwa
- **id()** - unikalny identyfikator
- **source()** - źródło danych
- **crs()** - układ współrzędnych (instancja klasy *QgsCoordinateReferenceSystem*)
- **extent()** - zasięg warstwy (*QgsRectangle*)
- **dataProvider()** - sterownik warstwy (*QgsVectorDataProvider* lub *QgsRasterDataProvider*)

QgsMapLayer



QgsProject

W uruchomionej aplikacji QGIS istnieje tylko jedna instancja klasy QgsProject. Dostęp do niej można uzyskać wywołując metodę *instance()*.

```
>>> QgsProject
<class 'qgis._core.QgsProject'>
>>> QgsProject.instance()
<qgis._core.QgsProject object at 0x...>
```

Lista wczytanych warstw:

```
>>> QgsProject.instance().mapLayers()
{'wektor_82a2e342_f2a0_4496_b5bc_fdd5a2bea4d8':
<qgis._core.QgsVectorLayer object at 0x…>,
'raster_d0c46c9f_2bd6_4745_9bbd_5bf265500431':
<qgis._core.QgsRasterLayer object at 0x…>, … }
```

Wyszukanie warstw po nazwie

Nazwy warstw nie są unikalne (mogą nazywać się tak samo) w związku z tym zwracana jest lista wszystkich znalezionych warstw. Jeśli nie znaleziono żadnej warstwy lista będzie pusta.

```
>>> QgsProject.instance().mapLayersByName('nazwa')
[<qgis._core.QgsVecotorLayer object at 0x…>, …]
```

Wyszukanie warstwy po id

ID warstwy jest unikalne, więc zwracana pojedyncza warstwa. Jeśli nie znaleziono warstwy funkcja zwróci *None*.

```
>>> QgsProject.instance().mapLayer('wektor_82a2e3…')
<qgis._core.QgsVecotorLayer object at 0x… >
```

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

W warstwach rastrowych kanały numerowane są kolejnymi liczbami naturalnymi, gdzie indeks 1 ma kanał pierwszy, 2 kanał drugi itd.

`raster` - instancja klasy **QgsRasterLayer**

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

- **sourceNoDataValue(kanal)** - liczba oznaczająca brak wartości rastra w danym kanale

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

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

Atrybuty klasy QgsRasterBandStats

- **elementCount** - liczba komórek (tylko z wartościami innymi niż brak wartości)
- **maximumValue** - maksymalna wartość
- **minimumValue** - minimalna wartość
- **mean** - średnia wartości
- **range** - zasięg wartości (max-min)
- **stdDev** - odchylenie standardowe
- **sum** - suma wartości

QgsRasterDataProvider.identify()

- **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 zwracane jest dla danego kanału *None*.

```
punkt = QgsPointXY(20, 50)
wartosc = raster.dataProvider().identify( punkt,
                  QgsRaster.IdentifyFormatValue )
print( wartosc.results() )
→ { 1 : 872.0, 2 : 304, 2 : None… }
```

QgsVectorLayer

- **geometryType()** - QgsWkbTypes.GeometryType
- **wkbType()** - QgsWkbTypes.Type

- **fields()** - schemat tabeli atrybutów (*QgsFields*)
- **featureCount()** - liczba obiektów na warstwie

- **getFeatures()** - iteracja po obiektach warstwy
- **getSelectedFeatures()** - iteracja po zaznaczonych obiektach
- **getFeature(int)** - pobranie pojedynczego obiektu o podanym id (*QgsFeature*)

`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 obiektach

`wektor` - instancja klasy **QgsVectorLayer**.

Iteracja po wszystkich obiektach:

```
for obiekt in wektor.getFeatures():
   print( obiekt )
```

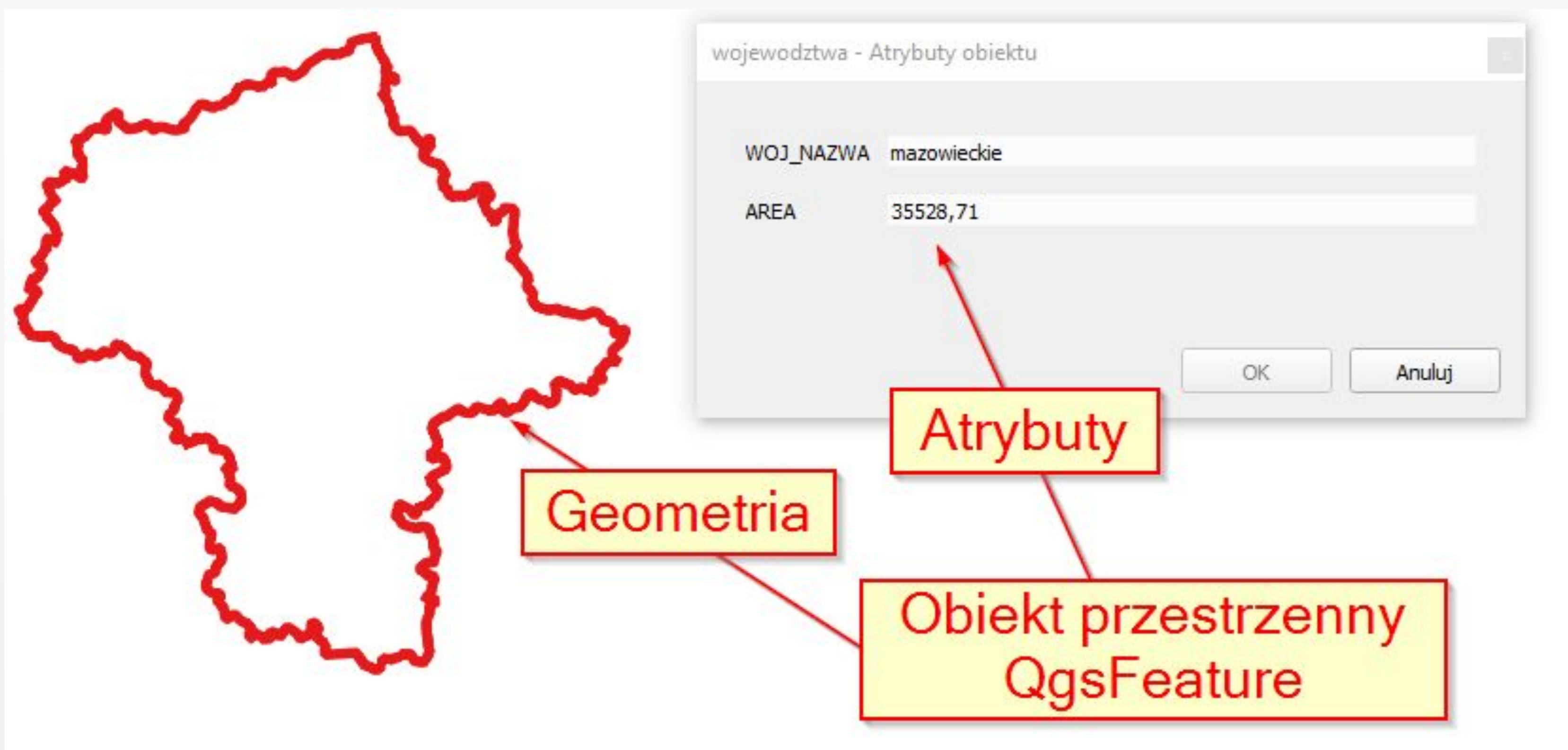
Iteracja po zaznaczonych obiektach:

```
for obiekt in wektor.getSelectedFeatures():
   print( obiekt )
```

Pobranie pojedynczego obiektu:

```
obiekt = wektor.getFeature( 0 )
```

Pojedynczy obiekt przestrzenny warstwy wektorowej reprezentowany jest klasą QgsFeature. Przechowuje on informacje o geometrii i wartościach atrybutów danego rekordu.



QgsFeature

- **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

Dostęp do wszystkich atrybutów:

```
for obiekt in warstwa.getFeatures():
   print( obiekt.attributes() )
→ [ 'tekst1', 19935.94 ]
→ [ 'tekst2', 17947.74 ]
```

Dostęp do wybranego atrybutu można wykonać na kilka sposobów

- wg indeksu kolumny

```
print( obiekt[1] )
→ 19935.94
```

- wg nazwy kolumny

```
print( obiekt['nazwa'] )
→ 'tekst1'
```

Podając w metodach *getFeatures* i *getSelectedFeatures* dodatkowy argument możemy filtrować zwracane obiekty z warstwy źródłowej:

- `getFeatures( [0, 1, 2] )` - obiekty o podanych id,
- `getFeatures( QgsRectangle(14, 49, 25, 55) )` - obiekty przecinające podany prostokąt,
- `getFeatures( ' "POLE" > 10 ' )` - obiekty spełniające dane wyrażenie,
- `getFeatures( QgsFeatureRequest(...) )` - zaawansowane filtrowanie za pomocą klasy *QgsFeatureRequest.*

Geometria obiektu

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)]]]	

- **Pojedynczy punkt**

```
QgsGeometry.fromPointXY( QgsPointXY(10, 21.5) )
```

- **Pojedyncza linia**

```
QgsGeometry.fromPolylineXY( [ QgsPointXY(0, 10),
QgsPointXY(11,16), …] )
```

- **Punkt wieloczęściowy**

```
QgsGeometry.fromMultiPointXY( [ QgsPointXY(0,
0), QgsPointXY(1,1), …] )
```

- **Z definicji WKT (Well-known text)**

```
QgsGeometry.fromWkt('POINT (0 0)')
```

`geometria` - instancja klasy **QgsGeometry**

- **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), …]
```

- **WKT**

```
print( geometry.asWkt() )
→ 'POINT (10 21.5)'
```

QgsWkbTypes

Klasa *QgsWkbTypes* zawiera informacje o wszystkich typach geometrii wspieranych przez QGIS. Atrybuty podzielone są na dwie grupy:

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

- **type()** - podstawowy typ geometrii (punkt, linia, poligon), *QgsWkbTypes.GeometryType*

```
>>> geometria.type()==QgsWkbTypes.PointGeometry
True
```

- **wkbType()** - szczegółowy typ geometrii, *QgsWkbTypes.Type*

```
>>> geometria.wkbType() == QgsWkbTypes.Point
True
```

```
>>> geometria.wkbType()==QgsWkbTypes.MultiPoint
False
```

- **isMultipart()** - czy obiekt jest wieloczęściowy, zwraca wartość logiczną

```
>>> geometria.isMultipart()
False
```

- **length()** - obwód lub długość (linie i poligony)

```
>>> geometria.length()
62.23
```

- **area()** - pole powierzchni (poligony)

```
>>> geometria.area()
6272.23
```

Powierzchnia i długość są zwracane w jednostkach układu współrzędnych.

Przekształcenia zwracają nowe geometrie w formie instancji klasy *QgsGeometry*:

- **buffer(size, segmenty)** - tworzenie bufora, wartość `segmenty` określa poziom zaokrąglania krzywych

```
>>> geometria.buffer( 1000, 5 )
<QgsGeometry: Polygon ((516819.765955
507565.867527,... >
```

- **centroid()** - środek masy, może znajdować się poza geometrią źródłową

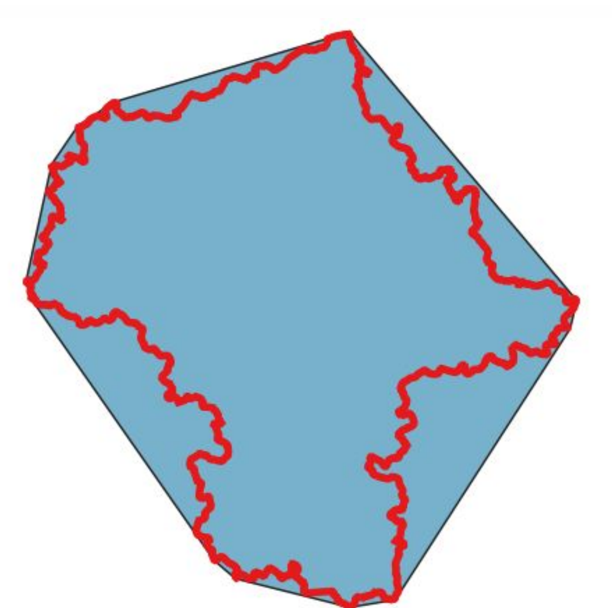
```
>>> geometria.centroid()
<QgsGeometry: Point (643023.1806 499325.4674)>
```

- **pointOnSurface()** - punkt wewnątrz geometrii

```
>>> geometria.pointOnSurface()
<QgsGeometry: Point (670544.8102 489861.4998)>
```

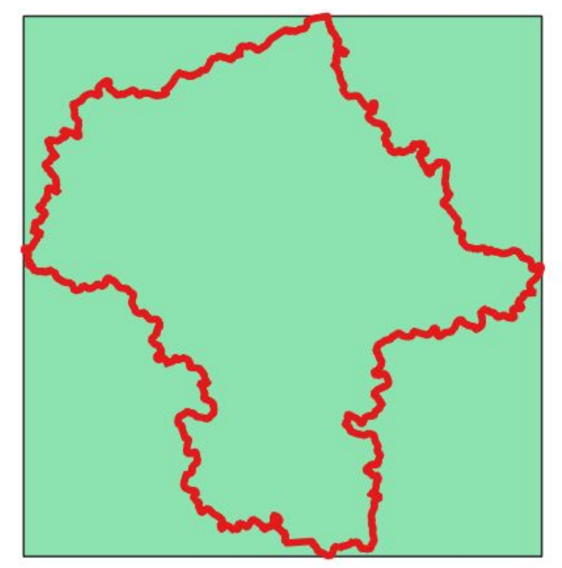
- **convexHull()** - otoczka wypukła

```
>>> geometria.convexHull()
<QgsGeometry: Polygon
((672758.0036 352477.8594,... >
```



- **boundingBox()** - prostokąt ograniczający geometrię (zwraca instancję klasy *QgsRectangle*)

```
>>> geometria.boundingBox()
<QgsRectangle: 517613.8493
352477.8594, 781450.3591
627244.5402>
```

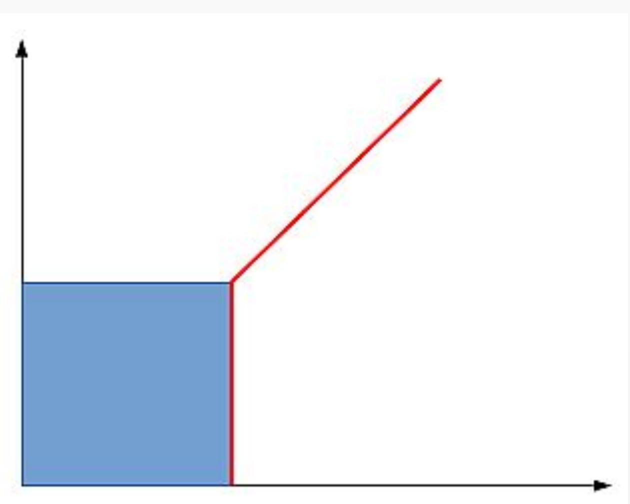


Zapytania przestrzenne

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

```
poligon = QgsGeometry.fromPolygonXY( [[ QgsPointXY(0,0),
QgsPointXY(0,1), QgsPointXY(1, 1), QgsPointXY(1,0) ]] )

linia = QgsGeometry.fromPolylineXY( [ QgsPointXY(1, 0),
QgsPointXY(1, 1), QgsPointXY(2, 2) ] )
```



```
linia.intersects( poligon )
```
→ **True**

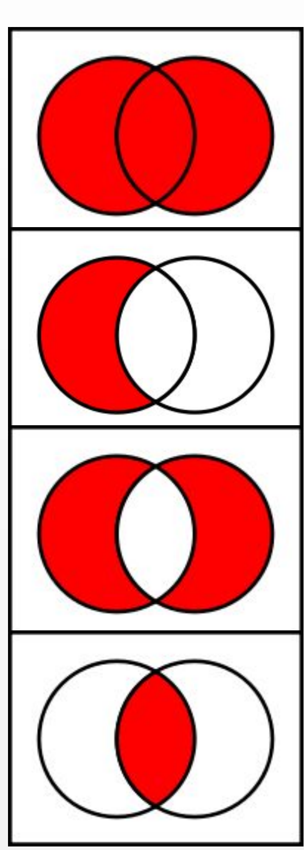
```
poligon.contains( linia )
```
→ **False**

```
poligon.overlaps( linia )
```
→ **False**

```
linia.touches( poligon )
```
→ **True**

```
linia.crosses( poligon )
```
→ **False**

- **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:

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

```
wgs84 = QgsCoordinateReferenceSystem.fromEpsgId(4326)
puwg92 = QgsCoordinateReferenceSystem('EPSG:2180')
ct = QgsCoordinateTransform(wgs84, puwg92,
QgsProject.instance())
```

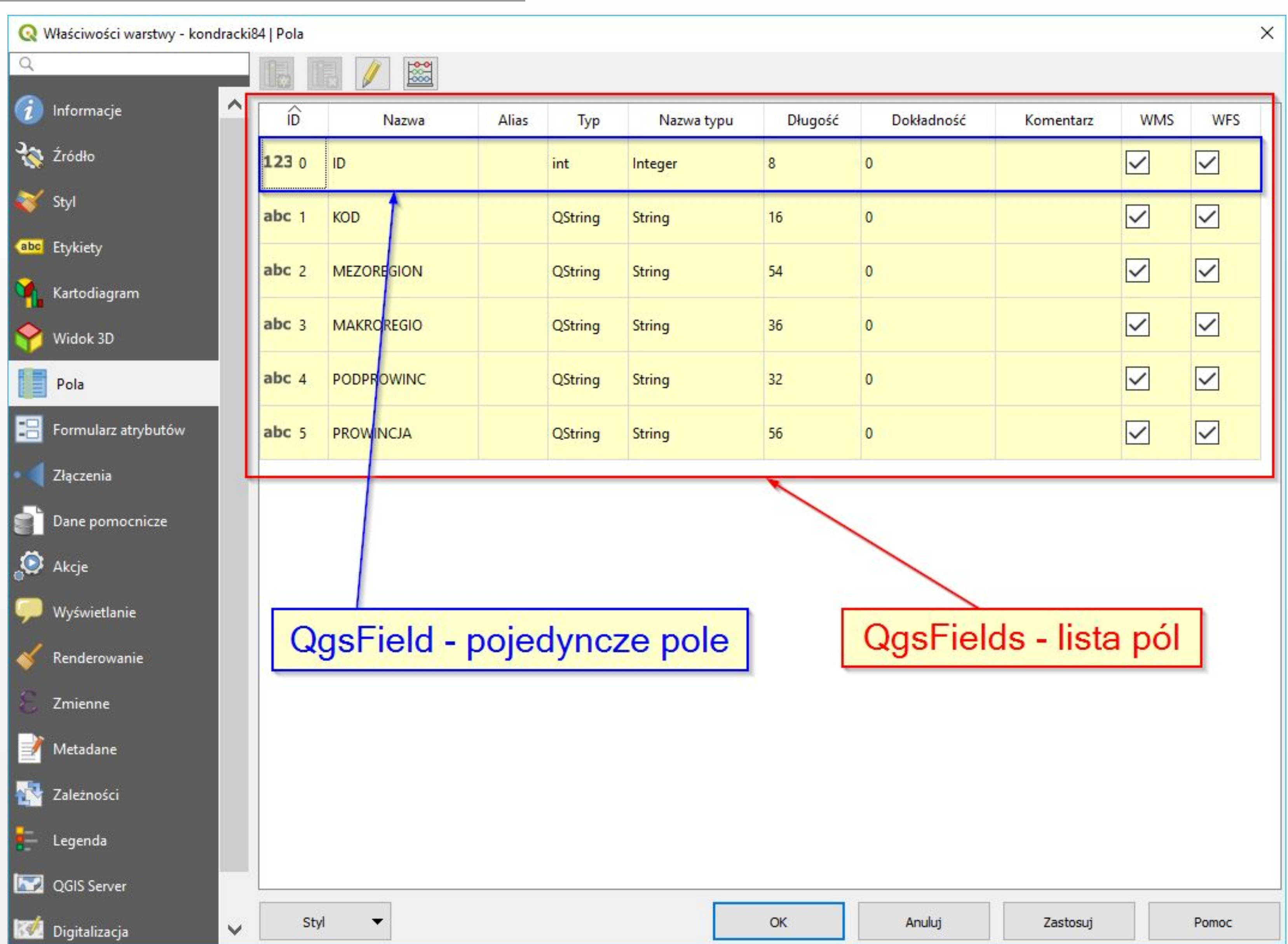
Transformacja klasy QgsPointXY

```
punkt_wgs = QgsPointXY(20, 50)
#Utworzenie nowego punktu
punkt_puwg = ct.transform( punkt_wgs )
```

Transformacja klasy QgsGeometry

```
geometry = QgsGeometry.fromPoint( punkt_wgs )
#Modyfikacja obiektu geometry
geometry.transform( ct )
```

Tabela atrybutów



QgsFields

`tabela` - instancja klasy **QgsFields**

- **count()** - liczba kolumn

```
>>> tabela.count()
6
```

- **field(int), field(str)** - pobranie pola wg indeksu (zaczynając od 0) lub nazwy, zwraca pole jako instancję klasy *QgsField*

```
>>> tabela.field( 0 )
<QgsField: KOD (String)>
>>> tabela.field( ‘Prowincja’ )
<QgsField: PROWINCJA (String)>
```

- **indexFromName(str)** - zwraca indeks pola o podanej nazwie, -1 jeśli nic nie znaleziono

```
>>> tabela.indexFromName( ‘KOD’ )
0
```

- **names()** - lista nazw pól

```
>>> tabela.names()
['ID', 'KOD', 'MEZOREGION', 'MAKROREGIO', 'PODPROWINC',
'PROWINCJA']
```

Modyfikacja schematu tabeli atrybutów:

- **append(QgsField)** - dodanie nowego pola
- **extend(QgsFields)** - rozszerzenie tabeli o pola z innej tabeli
- **remove(int)** - usunięcie pola wg indeksu

QgsField

`kolumna` - instancja klasy **QgsField**

- **name()** - nazwa pola

```
>>> kolumna.name()
'KOD'
```

- **length()** - długość pola (ilość znaków tekstu lub cyfr w liczbach)

```
>>> kolumna.length()
16
```

- **precision()** - dokładność liczb rzeczywistych (ilość cyfr po przecinku)
- **type()** - typ pola, zwracany jako wartość z klasy `QVariant`

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 pól

```
kolumna_id = QgsField( 'id', QVariant.Int )
tabela.append( kolumna_id )

kolumna_opis = QgsField('opis',QVariant.String)
tabela.append( kolumna_opis )
```

Usunięcie pól

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

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

```
from qgis.PyQt.QtCore import QVariant
kolumna = QgsField('Nazwa',QVariant.String)
kolumna.type() == QVariant.String
→ True
```

Typy danych wykorzystywane przez QGIS:

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

Stworzenie nowego obiektu.
Podanie listy pól z warstwy jest wymagane tylko w przypadku późniejszego przypisywania wartości atrybutów wg indeksu lub nazwy.

```
obiekt = QgsFeature( tabela )
```

Określenie geometrii i wartości atrybutów

```
geometria = QgsGeometry.fromPointXY(
                                QgsPointXY(20, 51) )
obiekt.setGeometry( geometria )
```

Przypisanie wartości wg indeksu pola

```
obiekt[1] = ‘Tekst’
obiekt.setAttribute(1, ‘Tekst’)
```

Przypisanie wartości wg nazwy pola

```
obiekt[‘id’] = 10
obiekt.setAttribute(‘id’, 10)
```

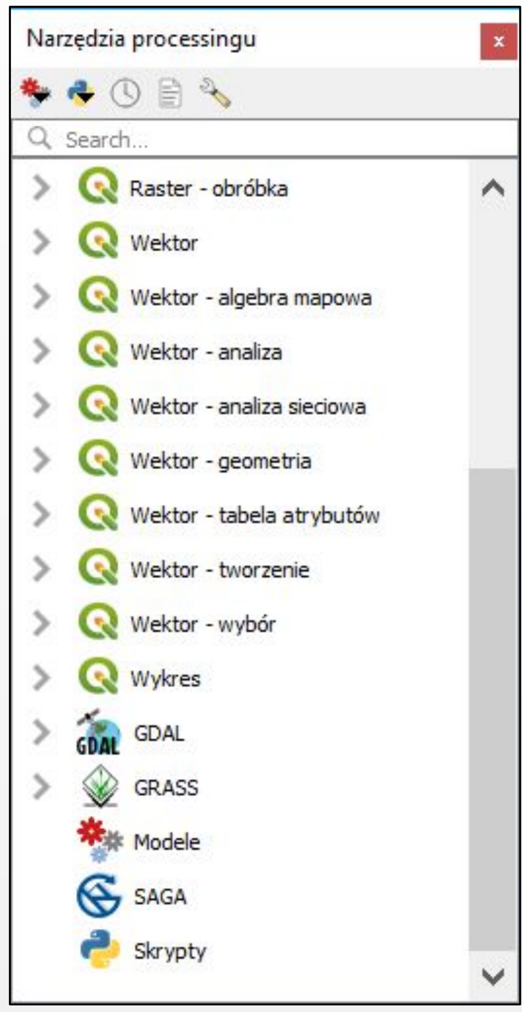
Określenie wielu atrybutów jednocześnie

Zadziała poprawnie jeśli liczba elementów i ich kolejność jest zgodna ze schematem tabeli atrybutów.

```
obiekt.setAttributes([20, ‘Opis’])
```

Narzędzia geoprocessingu i skrypty w języku programowania Python

Narzędzia geoprocessingu



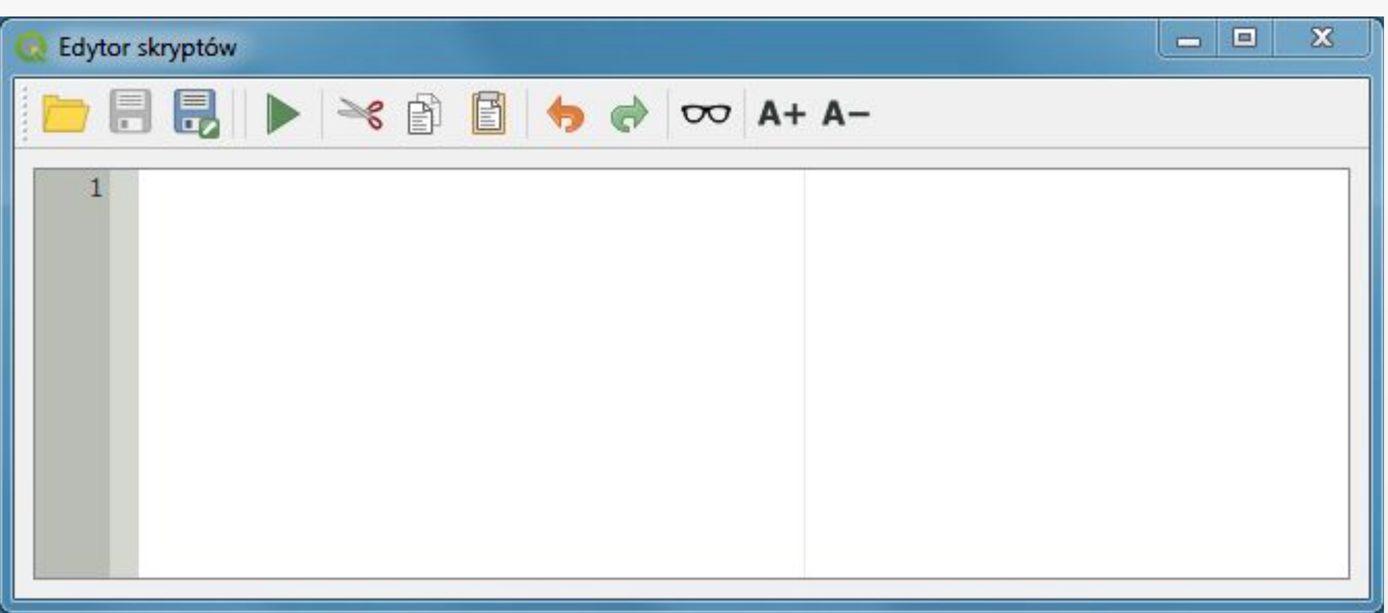
Narzędzia geoprocessingu zawierają algorytmy do tworzenia, zarządzania i analizy danych przestrzennych. Główne możliwości:

- wykonywanie algorytmów pojedynczo lub w trybie wsadowym,
- budowanie modeli na bazie dostępnych algorytmów,
- tworzenie własnych skryptów (algorytmów) w języku Python,
- możliwość dodawania kolejnych algorytmów z pomocą wtyczek.

Edytor dostępny jest w menu *Skrypty -> Create New Script...* lub *Create New Script from Template...*

Aby edytować istniejący skrypt, należy go odszukać na liście, kliknąć prawym klawiszem i wybrać opcję *Edit script*.

Edytor umożliwia szybkie testowanie algorytmu poprzez jego bezpośrednie uruchamianie.



- Skrypty Narzędzi geoprocessingu to pojedyncze pliki Pythona z roszerzeniem *.py*.
- Algorytm to specjalny obiekt, w którym zdefiniowane są wszystkie metadane (nazwa, opis, parametry wejściowe i wyjściowe) oraz funkcja wykonująca działanie.
- Każdy algorytm ma zdefiniowane parametry wejściowe i wyjściowe, które może ustawić użytkownik.
- Utworzone algorytmy mają takie same możliwości jak standardowo dostępne algorytmy i mogą być uruchamiane w trybie wsadowym lub wykorzystane przy tworzeniu modeli.

Algorytmy processingu można tworzyć na dwa sposoby:

1. Wykorzystując dekorator *@alg* - dostępne od QGIS 3.6
2. Tworząc nową klasę w oparciu o klasę bazową *QgsProcessingAlgorithm* - dostępne od QGIS 3.0

Pierwsze rozwiązanie jest zdecydowanie prostsze i szybsze, ponieważ skupiamy się na samym algorytmie. Nie jest też potrzebna szczegółowa wiedza dotycząca tworzenia własnych klas.

Dekorator to specjalny obiekt, który można przypisać do funkcji i wykonać dodatkowe operacje przed lub po jej wykonaniu. Dekoratory podaje się przed definicją funkcji z przedrostkiem @. Pojedyncza funkcja może mieć zdefiniowanych wiele dekoratorów.

```
@dekorator1
@dekorator2(argument)
@dekorator2.metoda(argument)
def funkcja():
    ...
```

Dekorator @*alg*

Dekorator @*alg* pozwala zmienić zwykłą funkcję Pythona w algorytm Narzędzi geoprocessingu. Służy on do:

- zdefiniowania metadanych algorytmu (nazwa, grupa),
- określenia parametrów wejściowych i wyjściowych,
- zarejestrowania algorytmu w Narzędziach geoprocessingu

Dekorator dostępny jest w module *qgis.processing*:

```
from qgis.processing import alg
```

Metadane określają nazwę i lokalizację algorytmu w panelu Narzędzi geoprocessingu. Ustawia się je podając dekorator @*alg* z poniższymi argumentami:

- **name** - unikalna nazwa algorytmu, która służy do jego identyfikacji, nie może zawierać znaków specjalnych, białych (spacja, tabulator itp.),
- **label** - wyświetlana nazwa algorytmu,
- **group** - unikalna nazwa grupy, nie może zawierać znaków specjalnych, białych (spacja, tabulator itp.),
- **group_label** - wyświetlana nazwa grupy, w której znajdować się będzie skrypt.

Każdy algorytm musi posiadać również dokumentację, którą umieszcza się jako komentarz wielolinijkowy pod definicją funkcji.

```
@alg(name='nazwa_systemowa',label='Nazwa wyświetlana',
     group='nazwa_grupy', group_label='Nazwa grupy')
def algorytm( … ):
    """ Dokumentacja algorytmu """
    …
```

- Istnieją dwie grupy parametrów:
  - `@alg.input`
  - `@alg.output`
- Każdy typ parametru jest zdefiniowany osobno i odpowiada konkretnej klasie z QGIS API
- Wszystkie parametry mają cztery podstawowe atrybuty, tylko *name* jest obligatoryjne:
  - `type` - typ parametru
  - `name` - unikalna nazwa identyfikująca parametr,
  - `label` - wyświetlany tekst parametru,
  - `default` - domyślna wartość atrybutu (można pominąć)
- Pozostałe argumenty uzależnione są od typu parametru (klasy QGIS API)

Parametry algorytmu

```
@alg.input( type=alg.SOURCE,
            name='WARSTWA_WEJSCIOWA',
            label='Warstwa wejściowa' )
@alg.input( type=alg.FIELD,
   name='POLE', label='Pole warstwy wejściowej',
   parentLayerParameterName='WARSTWA_WEJSCIOWA' )
@alg.input( type=alg.SINK,
            name='WARSTWA_WYJSCIOWA',
            label='Warstwa wyjściowa' )
@alg.output( type=alg.NUMBER,
             name='COUNT',
             label='Liczba obiektów' )
def algorytm( … ):
    …
```

@alg.input
Parametry wejściowe

Typ parametry	Klasa QGIS API	Opis
alg.MAPLAYER	QgsProcessingParameterMapLayer	Dowolna warstwa
alg.RASTER_LAYER	QgsProcessingParameterRasterLayer	Warstwa rastrowa
alg.VECTOR_LAYER	QgsProcessingParameterVectorLayer	Warstwa wektorowa
alg.SOURCE	QgsProcessingParameterFeatureSource	Źródło obiektów (warstwa wektorowa)
alg.MULTILAYER	QgsProcessingParameterMultipleLayers	Wiele warstw
alg.FIELD	QgsProcessingParameterField	Pole warstwy wektorowej
alg.EXTENT	QgsProcessingParameterExtent	Zasięg przestrzenny
alg.POINT	QgsProcessingParameterPoint	Punkt
alg.CRS	QgsProcessingParameterCrs	Układ współrzędnych
alg.BAND	QgsProcessingParameterBand	Kanał warstwy rastrowej

@alg.input
Parametry wejściowe

Typ parametry	**Klasa QGIS API**	**Opis**
alg.STRING	QgsProcessingParameterString	Tekst
alg.INT	QgsProcessingParameterNumber	Liczba całkowita
alg.NUMBER	QgsProcessingParameterNumber	Liczba rzeczywista
alg.RANGE	QgsProcessingParameterRange	Zasięg liczbowy
alg.DISTANCE	QgsProcessingParameterDistance	Odległość
alg.BOOL	QgsProcessingParameterBoolean	Wartość logiczna
alg.FILE	QgsProcessingParameterFile	Plik lub katalog
alg.ENUM	QgsProcessingParameterEnum	Lista wartości
alg.EXPRESSION	QgsProcessingParameterExpression	Wyrażenie

Parametry wyjściowe

Każdy algorytm musi mieć przynajmniej jeden parametr wyjściowy, z grupy *input* lub *output*. Parametry z grupy *output* nie są widoczne w oknie algorytmu, ale mogą być wykorzystane np. w modelarzu.

Dla każdego parametru wyjściowego funkcja główna musi zwracać wartość.

@alg.input
Parametry wyjściowe

Każdy algorytm musi mieć przynajmniej jeden parametr wyjściowy, z grupy *input* lub *output*.

Typ parametry	**Klasa QGIS API**	**Opis**
alg.SINK	QgsProcessingParameterFeatureSink	Warstwa wektorowa do zapisu obiektów
alg.VECTOR_LAYER_DEST	QgsProcessingParameterVectorDestination	Warstwa wektorowa
alg.RASTER_LAYER_DEST	QgsProcessingParameterRasterDestination	Warstwa rastrowa
alg.FILE_DEST	QgsProcessingParameterFileDestination	Plik
alg.FOLDER_DEST	QgsProcessingParameterFolderDestination	Katalog

@alg.output
Parametry wyjściowe

Typ parametry	Klasa QGIS API	Opis
alg.MAPLAYER	QgsProcessingOutputMapLayer	Dowolna warstwa
alg.RASTER_LAYER	QgsProcessingOutputRasterLayer	Warstwa rastrowa
alg.VECTOR_LAYER	QgsProcessingOutputVectorLayer	Warstwa wektorowa
alg.INT	QgsProcessingOutputNumber	Liczba całkowita
alg.NUMBER	QgsProcessingOutputNumber	Liczba rzeczywista
alg.DISTANCE	QgsProcessingOutputNumber	Dystans
alg.STRING	QgsProcessingOutputString	Tekst
alg.BOOL	QgsProcessingOutputBoolean	Wartość logiczna
alg.FILE	QgsProcessingOutputFile	Plik (utworzenie nowego)
alg.FOLDER	QgsProcessingOutputFolder	Katalog

Funkcja główna

```
@alg( … )
def algorytm( instance, parameters, context,
              feedback, inputs ):
   …
```

Główna funkcja musi przyjmować kilka argumentów, które są przekazywane w momencie uruchomienia algorytmu:

- **instance** - instancja klasy *QgsProcessingAlgorithm* pozwalająca na dostęp ustawionych przez użytkownika wartości poszczególnych parametrów,
- **parameters** - słownik ze zdefiniowanymi parametrami,
- **context** - kontekst w jakim uruchomiony jest algorytm, zawiera m.in. informacje o aktualnym projekcie QGIS,
- **feedback** - instancja klasy *QgsProcessingFeedback* pozwalająca na wyświetlanie komunikatów w trakcie wykonywania algorytmu,
- **inputs** - słownik zawierający wartości podane przez użytkownika, ale działa tylko dla parametrów numerycznych i tekstowych, dla pozostałych wartości są puste (*None*).

Obiekt *feedback*

```
@alg( … )
def algorytm( instance, parameters, context,
              feedback, inputs ):
    …
    feedback.pushInfo( 'Tekst do wyświetlenia' )
    feedback.setProgress( 50 )
    …
```

Informacje o postępie algorytmu

Metoda *processAlgorithm* otrzymuje jako czwarty parametr instancję klasy *QgsProcessingFeedback* (nazwany *feedback*), z pomocą której możliwe jest wyświetlanie informacji użytkownikowi, ustawianie postępu czy przerywanie działanian algorytmu. Główne metody obiektu:

- **setProgress(int)** – pozwala ustawić wartość paska postępu w dolnej części okna, parametr *int* powinien mieć wartość od 0 do 100,
- **setProgressText(str)** - ustawia tekst na pasku postępu,
- **pushInfo(str)** – wyświetla podany komunikat w oknie Log,
- **reportError(str, [bool=False])** – wyświetla informację o błędzie, jeśli drugi parametr jest ustawiony na **True** to skrypt zostanie zatrzymany,
- **isCanceled()** – zwraca informację czy użytkownik anulował skrypt i należy przerwać jego działanie (**True/False**).

Odczyt wartości parametrów

Aby uzyskać wartość parametru należy wywołać odpowiednią metodę z obiektu *instance* (klasa *QgsProcessingAlgorithm)*. Każdy parametr z grupy @alg.*input* ma własną funkcję rozpoczynającą się od *parametrAs...,* a kończącą nazwą parametru np. *parameterAsSource*. Metody zwracają wartości, których typ danych uzależniony jest od konkretnego parametru.

Każda metoda przyjmuje przynajmniej trzy parametry:

- **parameters** - słownik ze zdefiniowanymi parametrami,
- **name** - systemowa nazwa parametru,
- **context** - kontekst w jakim uruchamiany jest algorytm.

Jako *parameters* i *context* należy podać analogiczne obiekty przekazywane do funkcji głównej.

Odczyt wartości parametrów

```
def algorytm(instance, parameters, context, feedback, inputs):
    #Warstwa wektorowa (QgsProcessingFeatureSource)
    warstwa = instance.parameterAsSource( parameters,'nazwa1',
                                          context)

    #Pole tabeli, zwraca listę więc wybieramy pierwszy jej element
(QgsField)
    pole = instance.parameterAsFields( parameters,'nazwa2',
                                       context)[0]

    #Punkt (QgsPointXY), można określić układ współrzędnych
    punkt = instance.parameterAPoint( parameters, 'nazwa3',
                                      context, crs )

    #Układ współrzędnych (QgsCoordinateReferenceSystem)
    uklad = instance.parameterAsCrs( parameters, 'nazwa5',
                                     context)

    #Kanał rastra (int)
    kanal = instance.parameterAsInt( parameters, 'nazwa4',
                                     context )
```

Warstwa wyjściowa

```
def algorytm(instance, parameters, context, feedback, inputs):
    #Docelowa warstwa wektorowa
    (sink, dest_id) = instance.parameterAsSink(
        parameters, 'nazwa_systemowa', context,
        warstwa.fields(),    #Schemat tabeli atrybutów (QgsFields)
        warstwa.wkbType(),   #Typ geometrii (QgsWkbTypes.Type)
        warstwa.sourceCrs() #Układ współrzędnych
(QgsCoordinateReferenceSystem)
    )
    …

    #Dodane nowego obiektu
    sink.addFeature(nowy_obiekt, QgsFeatureSink.FastInsert)
    …
```

Wynik algorytmu

Jako wynik działania algorytmu należy zwrócić słownik. Każdy parametr wyjściowy stanowi jeden jego element, gdzie kluczem jest nazwa systemowa parametru, a wartością obiekt reprezentujący wynik.

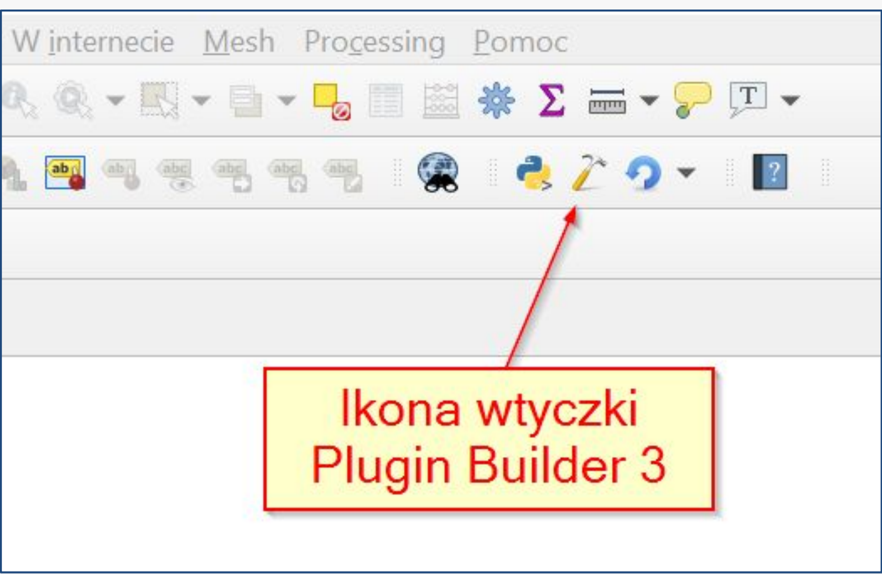
```
@alg.input( type=alg.SINK, name='WARSTWA', ...)
@alg.output( type=alg.NUMBER, name='LICZBA'... )

def algorytm(instance, parameters, context, feedback, inputs):
    …
    #Docelowa warstwa wektorowa
    (sink, dest_id) = instance.parameterAsSink(
       parameters, 'WARSTWA', context, ... )
    …

    #Zwrócenie wyniku działania algorytmu
    return {'WARSTWA': dest_id, 'LICZBA': 10}
```

Tworzenie wtyczek QGIS

Wtyczki służą do rozszerzenia standardowych możliwości danej aplikacji. QGIS wykorzystuje system wtyczek napisanych w językach C++ (wymagają kompilacji) 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*.

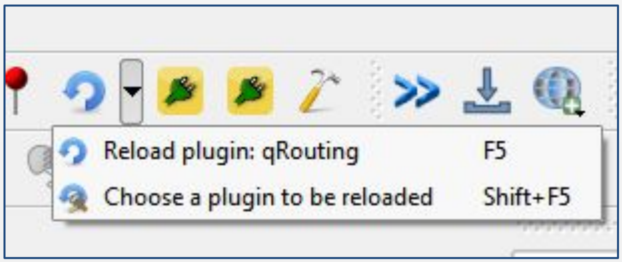
Wtyczki QGIS objęte są licencją GNU GPL w wersji 2.x lub nowszej. Rozpowszechniając wtyczkę autor zobowiązany jest dostarczyć użytkownikowi również jej kod źródłowy.

Narzędzie upraszczające tworzenie wtyczek poprzez wygenerowanie podstawowych plików i zasobów niezbędnych do stworzenia wtyczki.



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

Rozszerzenie oznaczone jest jako eksperymentalne, do jego instalacji należy włączyć pokazywanie eksperymentalnych wtyczek w opcjach 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 użytkownia 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.

- **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.

- **Internationalization** - tworzy pliki pomocne przy tworzeniu wtyczek wielojęzycznych.
- **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.

- **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 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 wtyczki.

Nową wtyczkę, należy skopiować do tego katalogu. Aby była ona widoczna w Menadżerze wtyczek należy zrestartować aplikację QGIS.

Dana wtyczka jest widoczna tylko w profilu, do którego została dodana. W pozostałych profilach należy ją zainstalować/skopiować indywidualnie.

Podstawowa struktura plików

metadata.txt → metadane wtyczki

__init__.py → plik podstawowy bibliotek Pythona, zawiera klasę *classFactory*, którą wykorzystuje QGIS do załadowania wtyczki

plugin.py → moduł zawierający główną klasę wtyczki

resources.qrc → zasoby wtyczki (głównie używane ikony)

resources.py → skompilowany plik zasobów .qrc

dialog.ui → okno dialogowe

dialog.py → skompilowane okno dialogowe

metadata.txt

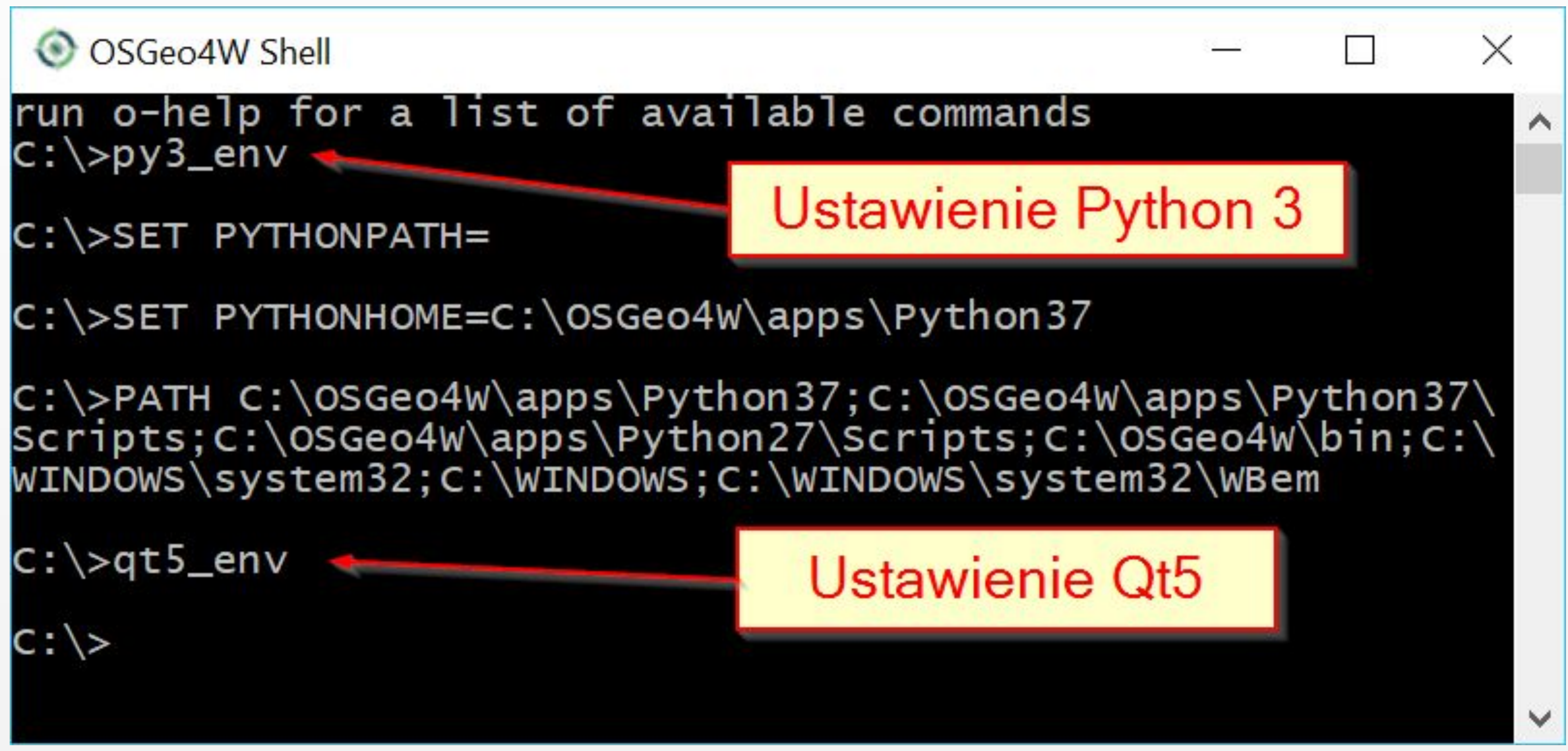
Plik w formacie INI zawierający metadane wtyczki.

Większość informacji, które tu się znajdują została wygenerowana automatycznie na podstawie danych wprowadzonych w Plugin Builder.

Instalując QGIS w wersji 3 mamy dostęp do specjalnej konsoli OSGeo4W Shell, w której ustawione są odpowiednie zmienne środowiskowe wymagane do poprawnej konfiguracji i uruchomienia dostarczonego interpretera Pythona. Często instalowane są dwie wersje Pythona 2 i 3 oraz Qt 4 i 5, aktualnie domyślnie uruchamiany jest Python2 i Qt4. Aby skorzystać z nowszych wersji należy po uruchomieniu konsoli wpisać polecenia:

```
> py3_env
> qt5_env
```

Python3 i Qt5 w OSGeo4W Shell



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

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

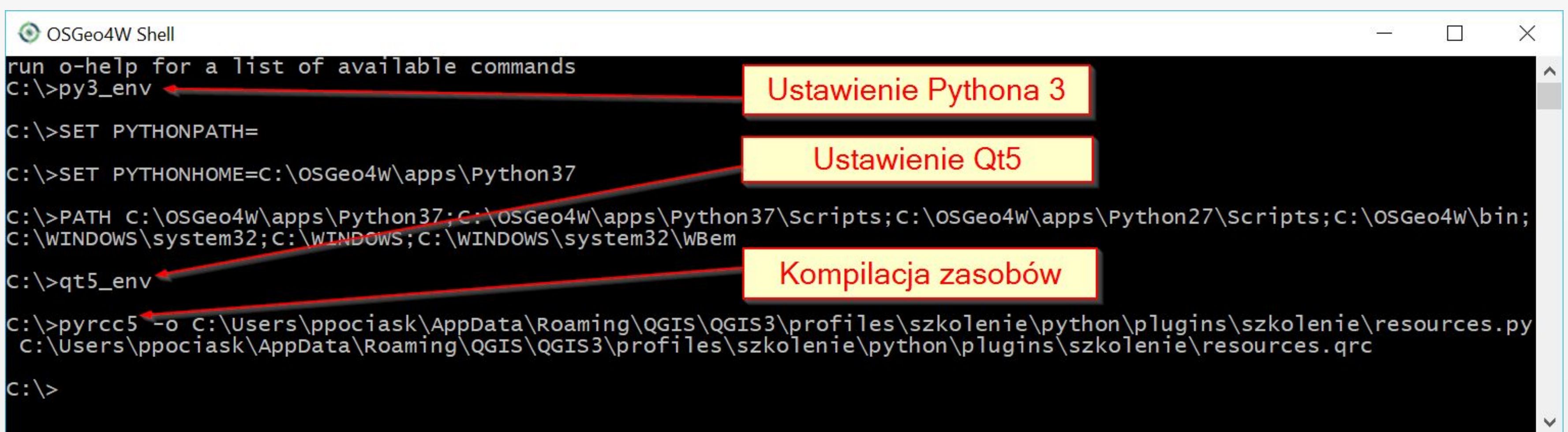
Aby wskazać zasób należy podać prefiks wraz z nazwą zasobu poprzedzone dwukropkiem.

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

Po zmianie zasobów lub edycji pliku należy go skompilować poleceniem *pyrcc5* 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.



Plik __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.

Klasa wtyczki

Główna klasa wtyczki znajduje się w pliku *nazwa_wtyczki.py*. Musi ona zawierać trzy metody wywoływane podczas ładowania lub wyłączania wtyczki.

Metoda ***__init__*** służy do stworzenia instancji klasy wtyczki w momencie jej uruchomienia i przekazaniu klasy QgisInterface.

Metoda ***initGui*** jest wywoływana w momencie włączenia wtyczki i służy do dodawania elementów interfejsu (przyciski, menu, panele), konfiguracji oraz rejestracji sygnałów i slotów.

Metoda ***unload*** jest wywoływana podczas wyłączania wtyczki i pozwala usunąć elementy interfejsu oraz rozłączyć istniejące sygnały i sloty.

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*:

```
self.add_action(
    icon_path,          #Ścieżka do pliku ikony
    text,               #Wyświetlany tekst
    callback,           #Funkcja wywoływana po wywołaniu akcji
    enabled_flag,       #akcja jest aktywna, domyślnie True
    add_to_menu,        #Ikona w menu, domyślnie True
    add_to_toolbar,     #Ikona na pasku narzędzi, domyślnie True
    status_tip,         #Tekst podpowiedzi w dymku
    whats_this,         #Tekst w pasku statusu
    parent              #Kontrolka rodzic dla akcji, domyślnie None
)
```

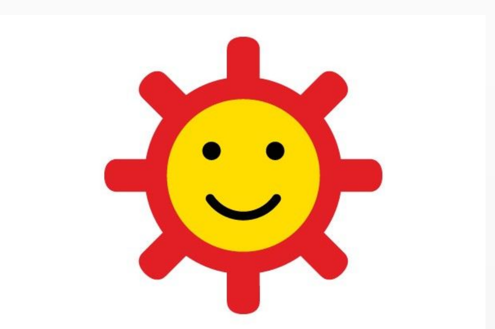
Qt i tworzenie interfejsu graficznego

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++ i Javy, a dzięki nakładkom (bindings) można korzystać z ich możliwości w wielu innych językach programowania. Dla Pythona istnieją dwie biblioteki umożliwiające korzystanie z Qt: PyQt i PySide.

Kto korzysta z Qt

https://en.wikipedia.org/wiki/Category:Software_that_uses_Qt



Główne moduły:

- **QtCore** – zawiera podstawowe klasy niegraficzne, mechanizm sygnałów i slotów, wyrażenia regularne itp.
- **QtGui** – zawiera klasy GUI służące do tworzenia okien dialogowych, obsługi kolorów i czcionek, rysowania 2D i 3D itp.
- **QtWidgets** - zbiór kontrolek (widżetów) do tworzenia interfejsu graficznego

Importowanie modułów

```
from qgis.PyQt.QtCore import *
from qgis.PyQt.QtGui import *
from qgis.PyQt.QtWidgets import *
```

zamiast *qgis.PyQt* możliwy jest import z modułu *PyQt5* np.

```
from PyQt5.QtWidgets import *
```

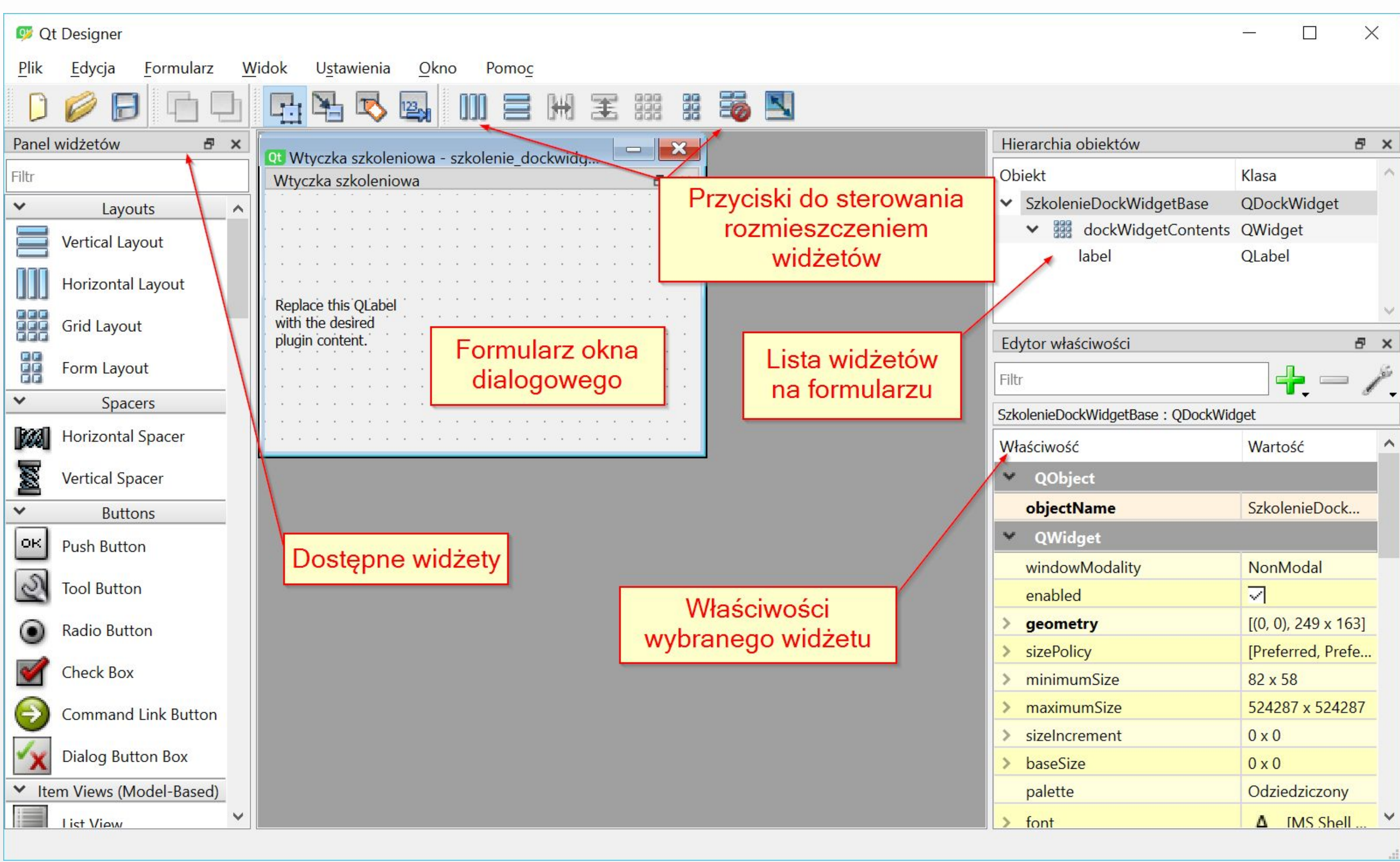
- **Qt Designer** - aplikacja graficzna do definiowania graficznego interfejsu użytkownika (okien dialogowych itp.)
- **Qt Linguist** - aplikacja wspomagająca tłumaczenie programu na różne języki
- **Qt Assistant** - aplikacja zawierająca rozbudowany system pomocy dla programistów
- **Qt Creator** - zintegrowane środowisko programistyczne.
- **uic** (User Interface Compiler) - kompilator plików *.ui zwykle generowanych za pośrednictwem programu Qt Designer (w PyQt odpowiednikiem tego narzędzia jest **pyuic4**)

Program *Qt Designer* pozwala wizualnie tworzyć okna dialogowe wykorzystywane przez pakiet Qt. Dane zapisywane są w plikach .ui (XML). Aby wykorzystać je w aplikacjach napisanych z pomocą PyQt należy je skompilować do plików .py narzędziem *pyuic4* lub bezpośrednio wczytać za pomocą funkcji *loadUiType*.

Qt Designer jest dostarczany razem z QGIS. W przypadku korzystania z instalatora *OSGeo4W* należy zaznaczyć bibliotekę *qt4-devel*.

W przypadku braku tej aplikacji można zainstalować pakiet *Qt Creator*.

Qt Designer



Okna dialogowe .ui

Okna dialogowe stworzone w programie *Qt Designer* zapisywane są w formacie UI (XML). Aby je wykorzystać w programach Python należy je skompilować do plików .py lub dynamicznie ładować w kodzie.

1. Wygenerowanie pliku .py w konsoli OSGeo Shell:

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

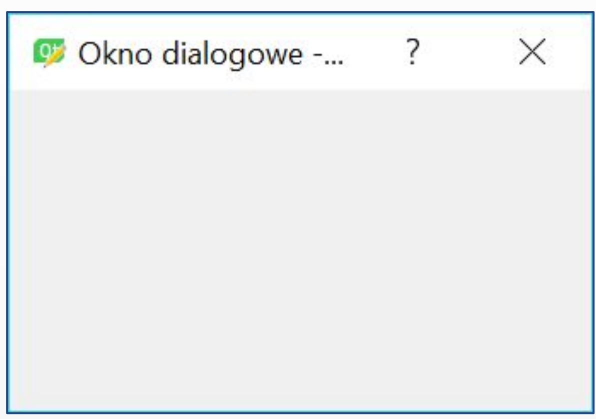
2. Dynamiczne ładowanie pliku .ui bez konieczności ręcznej kompilacji:

```
from qgis.PyQt import uic
from qgis.PyQt.QtWidgets import QDockWidget
import os

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

class PluginDockWidget(QDockWidget, FORM_CLASS):
    ...
```

Widżet (widget, kontrolka) – podstawowy element graficznego interfejsu użytkownika (np. okno, pole edycji, suwak, przycisk)

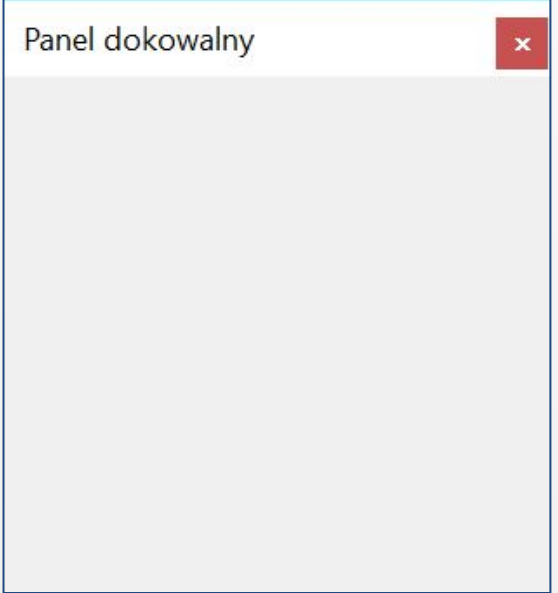
- **QDialog** – okno dialogowe,

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



- **QDockWidget** - panel dokowalny,

okno, które może zostać przyklejone do jednej z krawędzi okna głównego, panele można również dokować jeden na drugim.



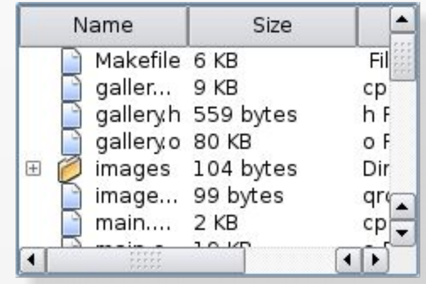
- **QPushButton** – przycisk
- **QLabel** – etykieta tekstowa
- **QLineEdit** – okienko do wpisywania tekstu (jednoliniowe)
- **QTextEdit** – okienko do wpisywania tekstu (wieloliniowe)
- **QRadioButton** – przycisk opcji
- **QCheckBox** – przycisk wyboru
- **QComboBox** – lista wyboru

- **QListWidget** – lista elementów

  QListWidgetItem – element listy

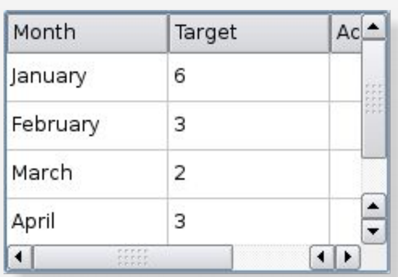
- **QTreeWidget** – lista elementów z widokiem drzewa

  QTreeWidgetItem – element listy z widokiem drzewa



- **QTableWidget** – tabela

  QTableWidgetItem – element tabeli (komórka)



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)
```

Sygnały i sloty

```
def drukuj():
   print( 'KLIK' )

#Stworzenie przycisku
przycisk = QPushButton( 'Kliknij mnie' )
#Po wywołaniu sygnału clicked zostanie wykonana funkcja
przycisk.clicked.connect(drukuj)
#Wyświetlenie przycisku
przycisk.show()
#Rozłączenie sygnału i slotu (funkcji)
przycisk.clicked.disconnect(drukuj)
```

QMessageBox

`QMessageBox` – okno komunikatu, można określić tytuł, treść i przyciski jakie zostaną wyświetlone.

```
from qgis.PyQt.QtWidgets import QMessageBox

msg = QMessageBox.question(None, 'Tytuł',
    'Czy zapisać zmiany?',
    QMessageBox.Yes | QMessageBox.No)


if msg == QMessageBox.Yes:
    #jeśli użytkownik wybrał Tak
    QMessageBox.information(None, 'Tak',
           'Zapisano zmiany.')

else:
    #jeśli użytkownik wybrał Nie
    QMessageBox.critical(None, 'Nie',
           'Odrzucono zmiany.')
```

Tytuł
Czy zapisać zmiany?
Tak Nie

Tak
Zapisano zmiany.
OK

Nie
Odrzucono zmiany.
OK

QInputDialog – okno pobrania danych

```
from qgis.PyQt.QtWidgets import QInputDialog
msg = QInputDialog.getText(None, 'Tytuł okna', 'Podaj login')


print( msg )
# ('Login', True)



if msg[1]:
    #użytkownik wprowadził wartość
else:
    #użytkownik kliknął Anuluj (Cancel)
```

QFileDialog

QFileDialog – okno wyboru pliku lub katalogu

```
from qgis.PyQt.QtWidgets import QFileDialog
plik, filtr = QFileDialog.getSaveFileName(None, 'Wybierz plik', 'C:/', 'Plik tekstowy (*.txt);;Obrazy (*.jpg *.png)')

print( plik, filtr )
# 'C:/plik.txt' 'Plik tekstowy (*.txt)


if plik:
    #użytkownik wybrał plik
else:
    #użytkownik kliknął Anuluj
```

Więcej informacji

http://www.qgis.org/api/ - oficjalna dokumentacja QGIS API (C++)

https://qgis.org/pyqgis/ - oficjalna dokumentacja QGIS API (Python)

https://docs.qgis.org/testing/en/docs/pyqgis_developer_cookbook/ - poradnik do PyQGIS, zawiera wiele przykładowych fragmentów kodu

http://docs.qgis.org/testing/en/docs/pyqgis_developer_cookbook/plugins.html - strona zawierająca podstawowe informacje o tworzeniu wtyczek.

http://osgeo-org.1560.x6.nabble.com/Quantum-GIS-f4099105.html - lista mailingowa QGIS

http://forum.grass-gis.pl/ - polskie forum QGIS

- **https://doc.qt.io/qt-5/** - oficjalna dokumentacja klas Qt (C++).
- **https://www.riverbankcomputing.com/static/Docs/PyQt5/** - dokumentacja dla biblioteki PyQt4.
- **http://www.python.rk.edu.pl/w/p/pyqt/** - polska strona o PyQt.
- **http://pyqt.sourceforge.net/Docs/PyQt4/new_style_signals_slots.html** - opis korzystania z sygnałów i slotów w PyQt.
- **http://pyqt.sourceforge.net/Docs/PyQt4/designer.html** - opis korzystania z plików wygenerowanych przez aplikację *Qt designer* w Python.

Dziękuję za uwagę