support

# Materiały szkoleniowe
# Python w QGIS
# (poziom średniozaawansowany)

MINISTERSTWO
KLIMATU

Sfinansowano ze środków
Narodowego Funduszu
Ochrony Środowiska
i Gospodarki Wodnej

# Spis treści:


**Biblioteki Python i repozytorium pypi.org** **3**
Publikacja bibliotek w repozytorium PyPI 3

**Omówienie budowy aplikacji QGIS** **6**
Struktura wewnętrzna 6
Konsola Python w QGIS 7
Konsola OSGeo4W Shell 8

**QGIS API** **11**
Biblioteki QGIS 13
Główne klasy QGIS API 13
Obiekt iface 14
Projekt QGIS 15

**Obsługa warstw przestrzennych** **17**
Dane rastrowe 18
Dane wektorowe 20
Obiekty przestrzenne 21
Geometria 22
Tworzenie geometrii 23
Informacje o geometrii 24
Typ geometrii 25
Zapytania przestrzenne 26
Tabela atrybutów 27
QgsFields 27
QgsField 28
Zarządzanie schematem tabeli atrybutów 28
Edycja 29
Stworzenie obiektu przestrzennego 29
Modyfikacja istniejącej warstwy 30

**Analizy przestrzenne** **33**
Analiza ciągów czasowych w NumPy 33
Trend i ekstrapolacja danych 34
Wizualizacja danych 36

**Wtyczki QGIS** **40**
Plugin Builder 40
Struktura wtyczki 45
Plik .ui 45
__init__.py 45
Plik .py 45
Plugin Reloader 46

**Framework Qt** **48**
Widżety 48
Qt Designer 50
Elementy interfejsu aplikacji Qt Designer 51
Układy (layouts) 51
Pliki .ui i Python 52
Sygnały i sloty 54
**Rozwijanie wtyczki** **54**
Akcje 54
Narzędzia mapy 56
QgsMapToolEmitPoint 56
Aktywacja narzędzi mapy 57
Akcje i narzędzia mapy 57
Dostęp do widżetów 59
QLineEdit 59
QSpinBox i QDoubleSpinBox 60
QgsMapLayerComboBox 60

# Biblioteki Python i repozytorium pypi.org

Biblioteka programistyczna to nazwa ogólna pliku lub plików, w których znajduje się kod realizujący jakieś funkcjonalności. Mogą się w niej znajdować podprogramy (funkcje, klasy), dane (zmienne) lub typy danych. Z tych elementów można korzystać w kodzie źródłowym pisanej aplikacji.
W języku Python biblioteką może być pojedynczy plik z rozszerzeniem .py (moduł) lub kilka plików zebranych w katalogu (paczka/pakiet). Aby skorzystać z zawartości biblioteki należy ją zaimportować za pomocą polecenia **`import`**.

**Przykładowe importy:**

```
# Import danych (zmiennej)
from math import pi

#Import podprogramu (funkcji)
from math import cos

#Import typu danych
from datetime import datetime
```

Python posiada bogatą bibliotekę modułów i paczek dostarczanych wraz ze standardowym interpreterem języka (polityka *Batteries included*). Dodatkowo możliwe jest instalowanie kolejnych rozszerzeń z oficjalnego repozytorium *Python Package Index* (https://pypi.org). Każdy programista może opublikować własne biblioteki, które następnie mogą być pobierane przez innych użytkowników. Najprostszym sposobem instalacji jest wykorzystanie narzędzia `pip`, które uruchamia się podając je jako parametr przy uruchamianiu interpretera z dodatkowym argumentem `-m`.

```
# Instalacja nowej biblioteki
python -m pip install <nazwa>

# Aktualizacja biblioteki
python -m pip install --upgrade <nazwa>

# Usunięcie biblioteki
python -m pip uninstall <nazwa>
```

## Publikacja bibliotek w repozytorium PyPI

Programiści mogą publikować własne biblioteki w repozytorium *Python Packege Index*. Najbardziej aktualny opis całego procesu opisany jest w oficjalnej dokumentacji i przed publikacją należy się z nim zapoznać:
https://packaging.python.org/tutorials/packaging-projects/

1. **Założenie konta**

Aby dodawać biblioteki należy zarejestrować się na stronie https://pypi.org/account/register. Podane dane będą potrzebne przy wgrywaniu bibliotek.

2. **Przygotowanie paczki**

Paczka zawierająca bibliotekę ma odpowiednią strukturę. Jest to katalog, w którym znajdują się pliki konfiguracyjne, metadane i sama biblioteka (w podkatalogu). Najważniejszym plikiem jest *setup.py*, czyli skrypt służący do budowania i instalacji biblioteki.

```
from distutils.core import setup, find_packages

setup(
      name='Nazwa_biblioteki',      # Musi być unikalna w całym repozytorium, może zawierać tylko litery, liczby i znaki _ oraz -
      version='1.0',                # Wersja
      author='Autor',               # Autor
      packages=find_packages(),     # Lista bibliotek do dołączenia, funkcja find_packages() wyszukuje je automatycznie w katalogu głównym paczki
      url='https://dokumentacja.pl',      # Strona domowa np. z dokumentacją
      license='LICENSE.txt',  # Plik z licencją
      description='Krótki opis biblioteki',
      long_description='Dłuższy opis biblioteki',     # Długi opis, najczęściej z pliku README.txt
)
```

Pełny i aktualny opis dostępnych argumentów funkcji **`setup`** można znaleźć na stronie https://setuptools.readthedocs.io/en/latest/setuptools.html#new-and-changed-setup-keywords.
Dodatkowo w paczce powinny znaleźć się pliki tekstowe:

- *LICENSE* zawierający licencję, na jakiej jest udostępniana biblioteka
- *README* z dłuższym opisem biblioteki, dobrze jeśli zawiera przykłady

3. **Budowanie paczki**

Mając paczkę ze wszystkimi plikami należy wygenerować plik do dystrybucji tzw. *Distribution Package*. Jest to archiwum zawierające paczkę, które można wgrać do repozytorium *PyPI* i wykorzystać do instalacji za pomocą polecenie *pip*.
W celu zbudowania paczki należy z wiersza poleceń przejść do katalogu głównego paczki i uruchomić polecenie:

```
python setup.py sdist bdist_wheel
```

To utworzy nowy podkatalog *dist*, a w nim dwa pliki z rozszerzeniami:

- **.tar.gz** - zawierający kod źródłowy biblioteki
- **.whl** - archiwum do dystrybucji paczki

4. **Upload biblioteki**

Do wgrywania gotowych paczek można wykorzystać narzędzie *twine*. Jest to biblioteka Pythona i można je zainstalować poprzez *pip*:

```
python -m pip install twine
```

Jeśli *twine* jest zainstalowany należy wejść do katalogu paczki i uruchomić polecenie:

```
python -m twine upload dist/*
```

Następnie należy podać dane autoryzacyjne do repozytorium PyPI.

Po wykonaniu powyższych czynności bibliotekę można zainstalować za pomocą polecenia *pip* jak każdą inną bibliotekę.

# Ćwiczenie

**Treść zadania**

Stwórz moduł o nazwie *analysis.py* i zdefiniuj w nim dwa importy: **`numpy`** oraz obiektu **`pyplot`** z biblioteki **`matplotlib`**.

**Opis**

Należy utworzyć pusty plik tekstowy o nazwie analysis.py oraz dodać dwa importy. Bibliotekę **`numpy`** należy zaimportować w całości, natomiast z biblioteki **`matplotlib`** można zaimportować bezpośrednio obiekt **`pyplot`**.

**Kod źródłowy**

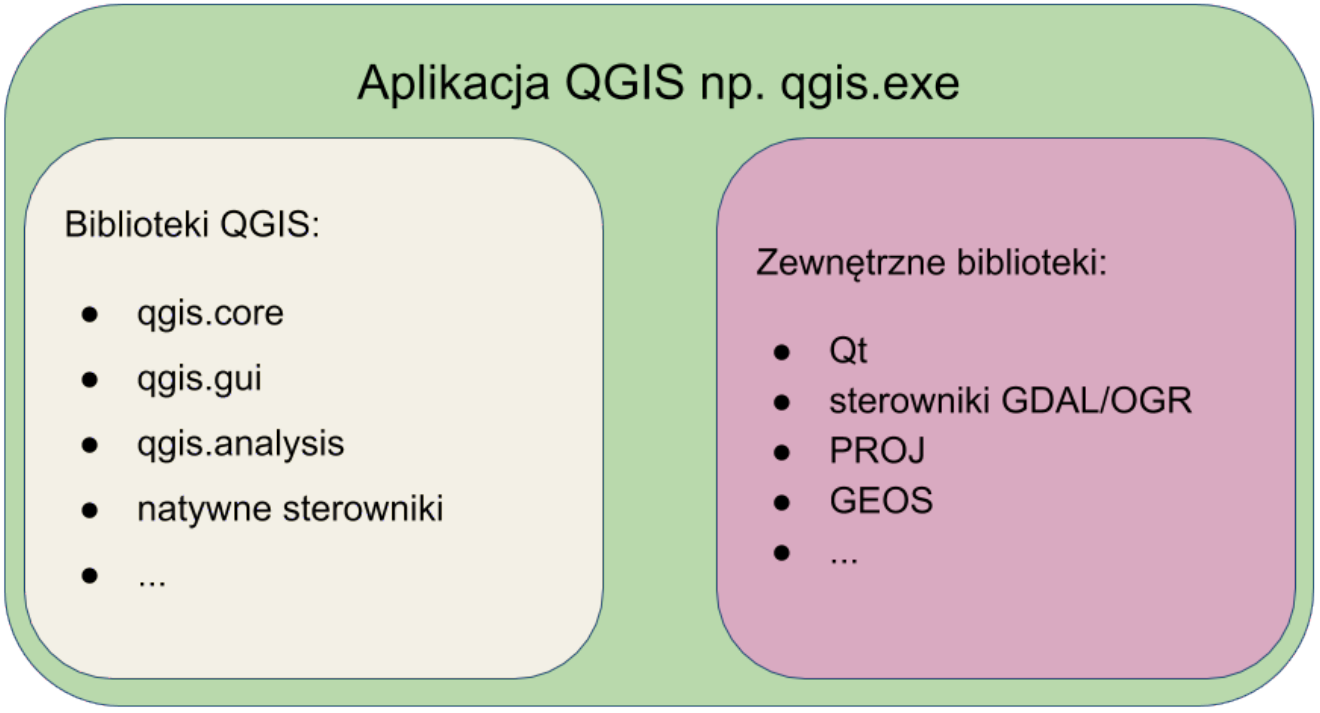
```
# Import biblioteki NumPy
import numpy
# Import obiektu do generowania wykresów
from matplotlib import pyplot
```

# Omówienie budowy aplikacji QGIS

## Struktura wewnętrzna

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

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




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

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

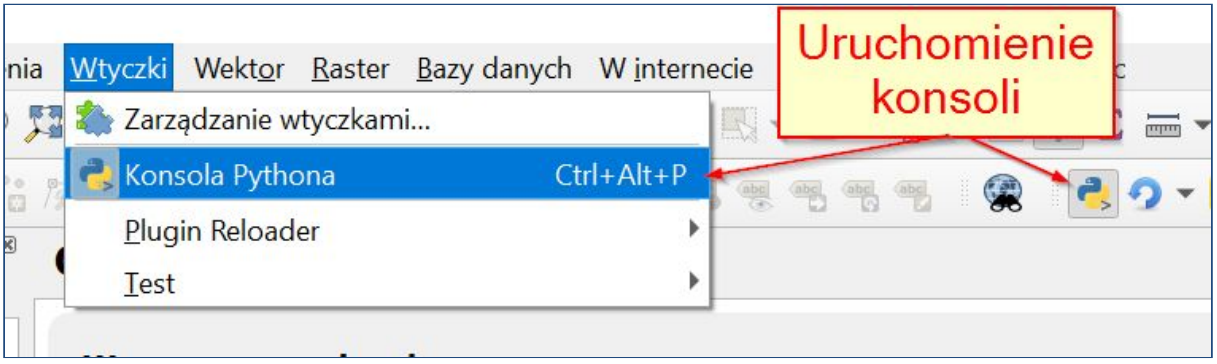
Są one wykorzystywane wewnętrznie w QGIS. Programując w Pythonie nie korzysta się z tych projektów bezpośrednio ale za pomocą bibliotek QGIS API. Szczegółowo zostaną one opisane w rozdziale *Biblioteki QGIS*.

# Konsola Python w QGIS

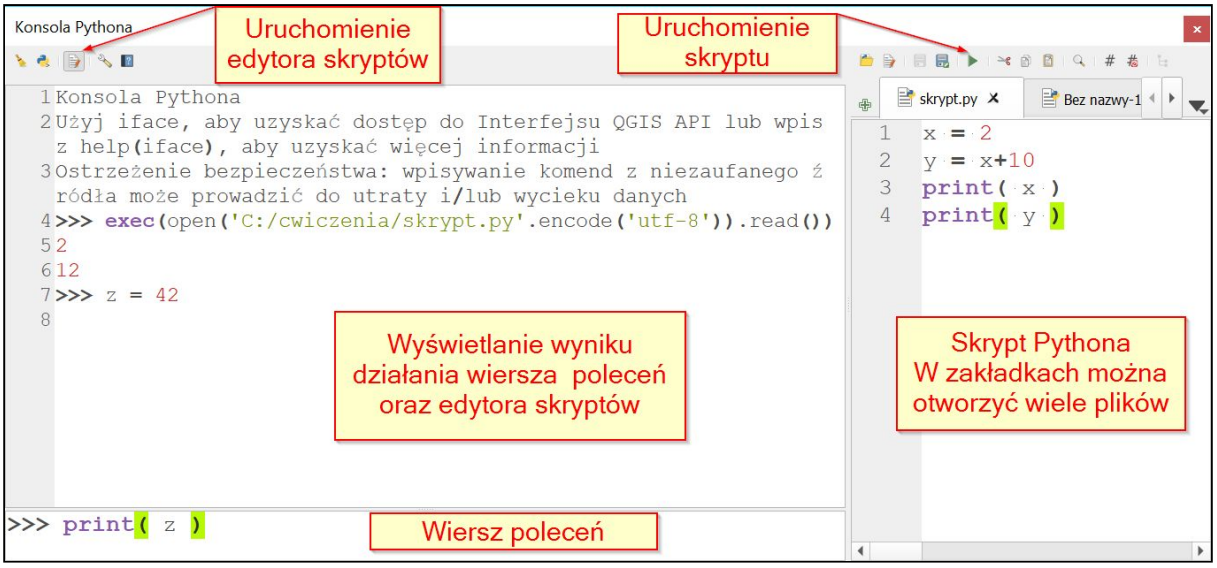
Aplikacja QGIS wykorzystuje Pythona jako język skryptowy. Wbudowana *Konsola Pythona* to nic innego jak nakładka graficzna na interpreter tego języka. Umożliwia ona pisanie kodu i wywoływanie go w interpreterze w dwóch trybach:

- 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.
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.

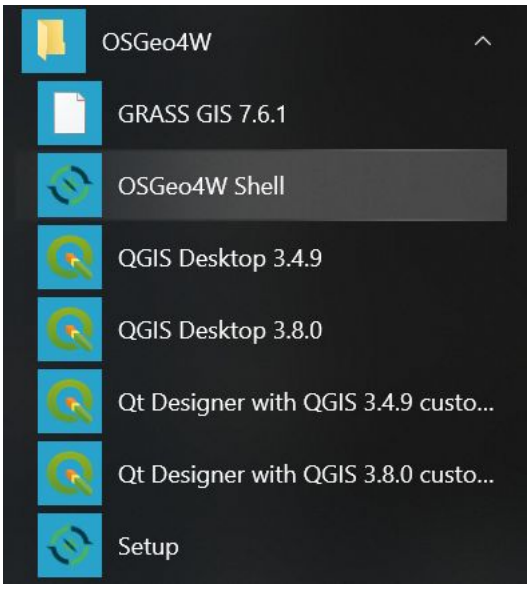



**Elementy Konsoli**

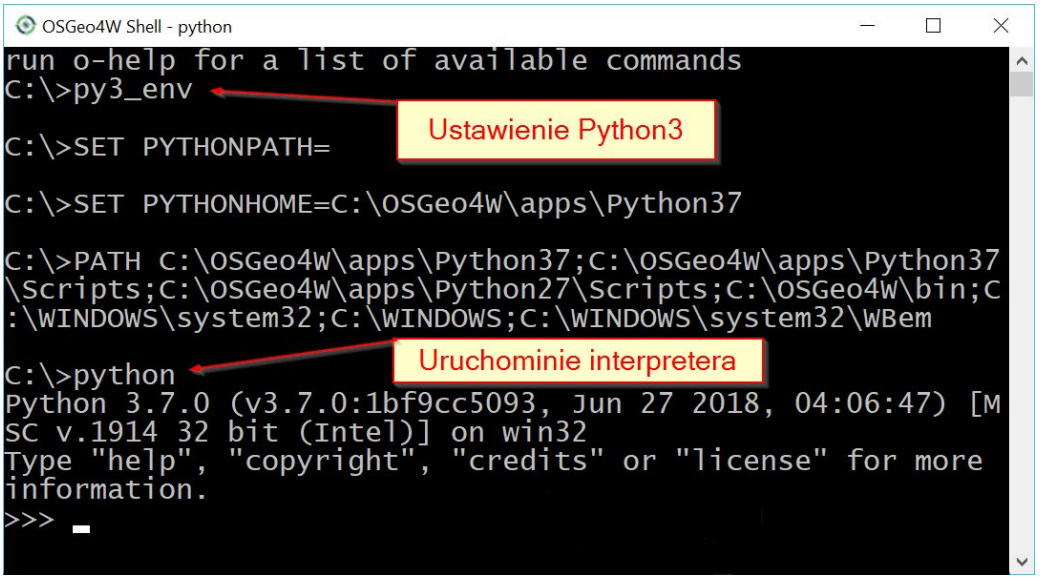


## Konsola OSGeo4W Shell

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




Domyślnie uruchamiany jest Python 2, aby skorzystać z nowszej wersji należy po uruchomieniu konsoli wpisać polecenie **`py3_env`**, a następnie **`python`**. Aby zamknąć interpreter Python należy wybrać kombinację klaswiszy Ctrl+Z i wcisnąć Enter.

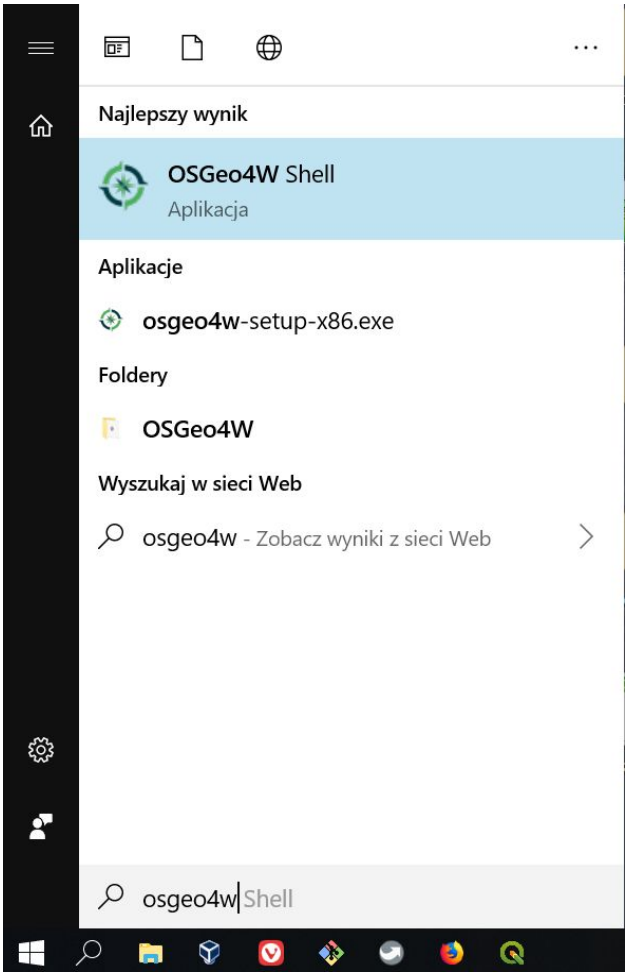



Dodatkowo warto później uruchomić polecenie **`qt5_env`** aby mieć dostęp do narzędzi Qt 5. Są one wykorzystywane przy tworzeniu wtyczek i szczegółowo zostaną omówione w rozdziale dotyczącym *Qt*.

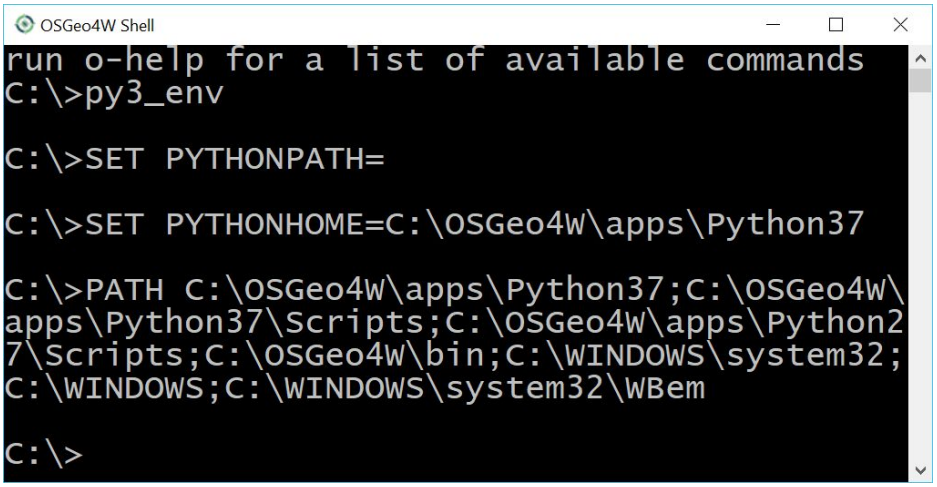
# Ćwiczenie

**Treść zadania**
Uruchom konsolę *OSGeo4W Shell* i uruchom interpreter Pythona 3.

**Opis**
Aby uruchomić konsolę *OSGeo4W Shell* należy w wyszukać odpowiednie polecenie w menu Start systemu Windows.




Pojawi się okno konsoli. Domyślnie ustawiony jest w niej Python w wersji 2. Aby przełączyć się na nowszą wersję należy wpisać polecenie `py3_env`. Po jego wpisaniu w konsoli pojawi się kilka wpisów informujących o zmianach w zmiennych środowiskowych.




Aby uruchomić interpreter Pythona należy wpisać polecenie `python`. Jeśli wszystko działa poprawnie w konsoli pojawi się informacja o wersji uruchomionego interpretera (powinna być większa niż 3.6), a kursor powinien migać w linijce rozpoczynającej się od znacznika `>>>`. Oznacza to gotowość do przyjmowania poleceń Pythona.

# QGIS API

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

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

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

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

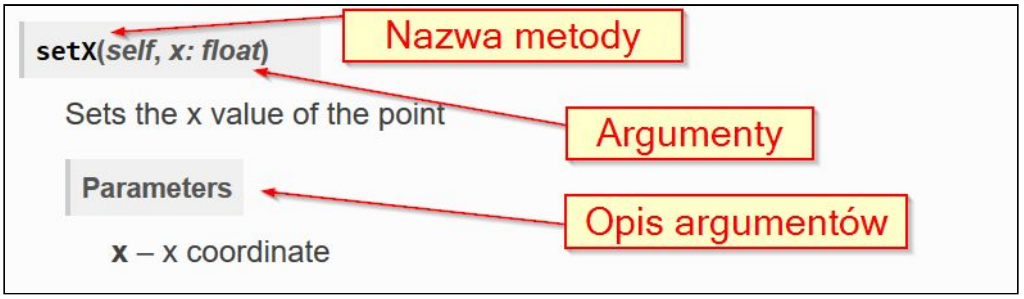
Strona główna zawiera wyszukiwarkę oraz spis wszystkich dostępnych klas z podziałem na moduły, w których się znajdują. Po wejściu w dokumentację danej klasy znajdziemy w niej krótki jej opis oraz spis metod (funkcji) i atrybutów (zmiennych) dostępnych z jej poziomu. Po kliknięciu w nazwę można przejść do szczegółowszych informacji dotyczących danego elementu klasy np. jakie argumenty przyjmuje metoda i co zwraca jako wynik działania.

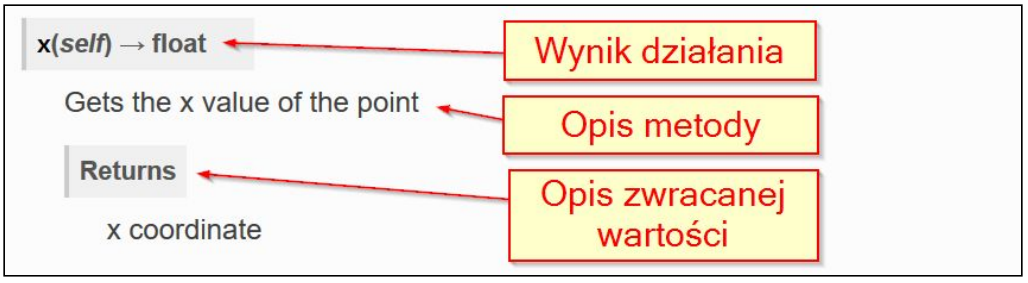
Przykład opisu klasy w dokumentacji QGIS API. **`QgsPointXY`** - klasa reprezentująca punkt w przestrzeni 2D:


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.

```
# Stworzenie instancji na podstawie współrzędnych
punkt1 = QgsPointXY( 10, 15 )
# Stworzenie instancji na podstawie istniejącej instancji tej samej
klasy (klonowanie), punkt2 będzie miał te same współrzędne co punkt1
punkt2 = QgsPointXY( punkt1 )
```

Opis metod w dokumentacji:



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

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

## Biblioteki QGIS

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

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

Import obiektów z powyższych klas można wykonać w następujący sposób:

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

## Główne klasy QGIS API

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

### qgis.core

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

- **`QgsWkbTypes`** - przechowuje informacje o typach geometrii
- **`QgsPointXY`** – klasa reprezentująca punkt 2D
- **`QgsRectangle`** - klasa reprezentująca prostokąt określony współrzędnymi min_x, min_y, max_x, max_y
- **`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
- **`QgsMapLayerComboBox`** - pole wyboru warstwy z listy wczytanych do QGIS, automatycznie aktualizowane przy dodawaniu/usuwaniu warstw.
- **`QgsMapTool`** - klasa bazowa dla narzędzi mapy QGIS,
- **`QgsMapToolEmitPoint`** - klasa reagująca na klikanie po mapie.

# Ćwiczenie

**Treść zadania**
Z poziomu *Konsoli Pythona* w QGIS zaimportuj klasę **`QgsPointXY`** z modułu **`qgis.core`**.

**Opis**
Po uruchomieniu *Konsoli Pythona* należy wpisać polecenie:

```
from qgis.core import QgsPointXY
```

i wcisnąć przycisk *Enter*.

## Obiekt *iface*

Obiekt **`iface`** jest instancją klasy **`QgisInterface`**. Jest to specjalny obiekt, który umożliwia z poziomu języka Python na sterowanie oknem uruchomionej aplikacji QGIS. Dzięki niej możliwe jest m.in. wczytywanie warstw, pobranie warstwy aktywnej, dodanie paneli dokowanych, rejestracja nowych pasków narzędzi wraz z przyciskami czy dodawanie menu.
Istnieje tylko jedna instancja klasy **`QgisInterface`**, nie jest możliwe jej utworzenie, a jedynie pozyskanie z QGIS. Jest on domyślnie dostępny w *Konsoli Pythona* oraz we wtyczkach, ale możliwe jest jego zaimportowanie w dowolnym miejscu:

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

Wybrane metody obiektu **`iface`**:

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

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

# Ćwiczenie

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

**Opis**
Główne okno mapy QGIS to instancja klasy **`QgsMapCanvas`**. Aby ją zwrócić należy skorzystać z obiektu **`iface`** i jego metody **`mapCanvas()`**. W dokumentacji należy zlokalizować metodę, która zwraca zasięg widoku mapy. Jest to metoda **`extent`**, która nie przyjmuje żadnych argumentów. Wynikiem jest klasa **`QgsRectangle`** reprezentująca prostokątny obszar mapy ze współrzędnymi minimalnymi (dolny lewy wierzchołek) i maksymalnymi (górny prawy wierzchołek).

**Kod źródłowy**

```
iface.mapCanvas().extent()
# <QgsRectangle: -160372.58668964554090053 258820.63609923399053514,
1295374.55022055562585592 746699.75786408013664186>
```

## Projekt QGIS

Informacje o projekcie QGIS są przechowywane w klasie **`QgsProject`**. Z jej poziomu możliwe jest m.in. wyszukiwanie i modyfikowanie warstw w legendzie czy pozyskanie informacji o głównym układzie współrzędnych.
Podobnie jak w przypadku klasy **`QgisInterface`** QGIS posiada pojedynczą instancję tej klasy reprezentującą aktualnie wczytany projekt. Jednak dostęp do tej instancji jest nieco inny:

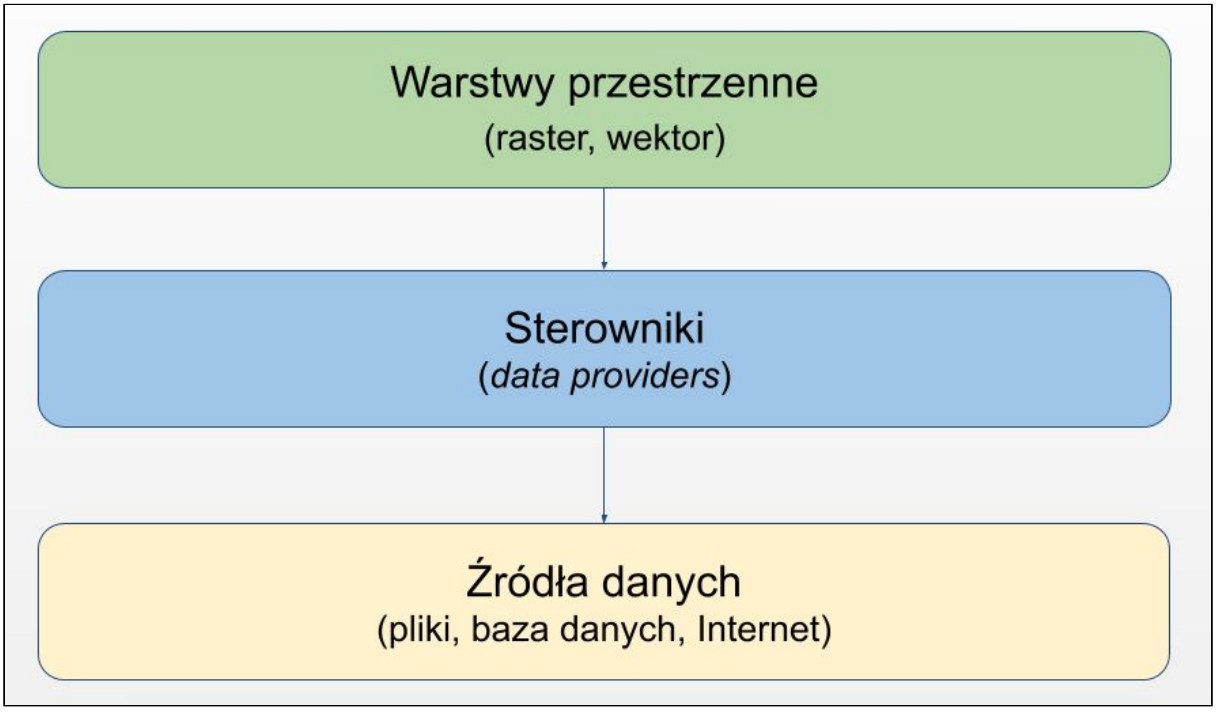
```
# Import jest wymagany wszędzie poza Konsolą Pythona
from qgis.core import QgsProject

# Pobranie instancji klasy QgsProject reprezentującej aktywny projekt
projekt = QgsProject.instance()
```

```
# Układ współrzędnych projektu
projekt.crs()
# <QgsCoordinateReferenceSystem: EPSG:2180>
```

# Obsługa warstw przestrzennych

QGIS wspiera wiele formatów danych przestrzennych. Do ich obsługi wykorzystywane są tzw. sterowniki (*data providers*), które odpowiadają za komunikację ze źródłem danych (plikiem, bazą danych). Sterowniki odczytując informacje ze źródła konwertują je do ujednoliconej formy warstw przestrzennych QGIS, dzięki temu niezależnie od formatu każda warstwa jest obsługiwana przez QGIS w ten sam sposób. Odpowiadają one również za zapisywanie zmian.

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




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

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

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

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

## Dane rastrowe

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

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

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

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

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

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

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

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

```
# Sterownik warstwy rastrowej (QgsRasterDataProvider)
sterownik = raster.dataProvider()
# Obliczenie statystyk kanału pierwszego
statystyki = sterownik.bandStatistics( 1 )
# mean - wartość średnia wszystkich komórek rastra
print( statystyki.mean )
# 173
```

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

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

# Ćwiczenie

**Treść zadania**

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.

**Opis**

Na początku należy utworzyć definicję funkcji, która przyjmuje argumenty **`raster`** i **`point`**. Do odczytu wartości rastra w podany, 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.

**Kod źródłowy**

```
#Import użytej klasy
from qgis.core import QgsRaster

def raster_identify( raster, point ):
      # Sterownik warstwy rastrowej (QgsRasterDataProvider)
      provider = raster.dataProvider()
      # Odczyt wartości komórek w podanym punkcie
      data = provider.identify( point, QgsRaster.IdentifyFormatValue )
      # Pobranie słownika z wartościami
      result = data.results()
      # Lista kanałów rastra
      bands = []
      # Lista wartości rastra
      values = []
      #Iteracja po kanałach, sortujemy klucze słownika, aby mieć pewność,
      # że zostaną zwrócone w rosnącej kolejności 1, 2, 3, itd.
      for band in sorted(result):
            # Dodanie kanału do listy
            bands.append( band )
            # Dodanie wartości do listy
            values.append( result[band] )
      # Zwrócenie obu list
      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`**)
- **`featureCount()`** - liczba obiektów na warstwie
- **`getFeatures()`** - iteracja po obiektach warstwy
- **`getSelectedFeatures()`** - iteracja po zaznaczonych obiektach

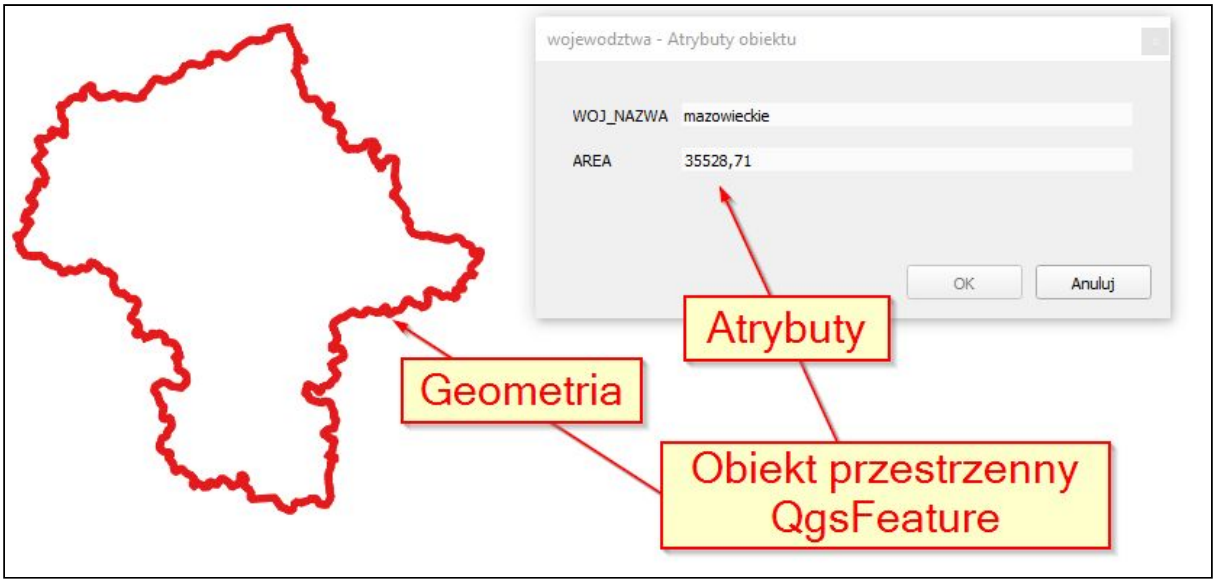
- **`getFeature( int )`** - pobranie pojedynczego obiektu o podanym id (**`QgsFeature`**)

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

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

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

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

# Pobranie pojedynczego obiektu:
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 ]
```

## 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.

### Tworzenie geometrii

Nowe geometrie można tworzyć na kilka sposobów:

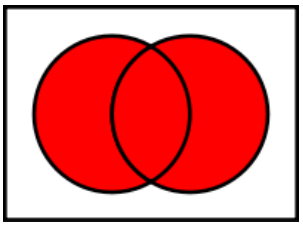
- z listy wierzchołków (zgodnie z powyższą tabelą) - służą do tego metody rozpoczynające się od przedrostka **`from`**, które są wywoływane bezpośrednio z klasy **`QgsGeometry`**:

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

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

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

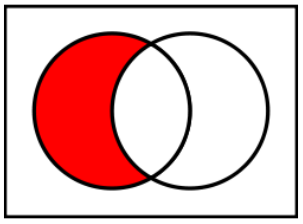
- przekształcając istniejące obiekty - metody zwracające nową instancję klasy **`QgsGeometry`**:

```
# 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()
```

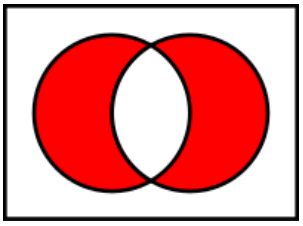
- wykonanie jednej z operacji przestrzennych na dwóch geometriach:
  - **`combine`** - połączenie dwóch geometrii w jedną,



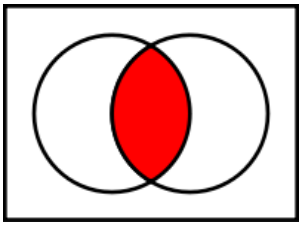
- **`difference`** - zwraca geometrię, która nie jest wspólna z drugą,



- **`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:

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

### Informacje o geometrii

Metody, które zwracają informacje o danej geometrii:

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

# Obwód lub długość (linie i poligony)
print( geometria.length() )
# 62.23

# Pole powierzchni (poligony)
print( geometria.area() )
# 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

**Treść zadania**
Wykonaj pętlę na obiektach warstwy *miasta* i wydrukuj wierzchołki tworzące geometrię tych obiektów.

**Opis**
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 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 miasta jest punktowa więc właściwą metodą jest **`asPoint()`**. Wynik działania tej metody należy wydrukować funkcją **`print()`**.

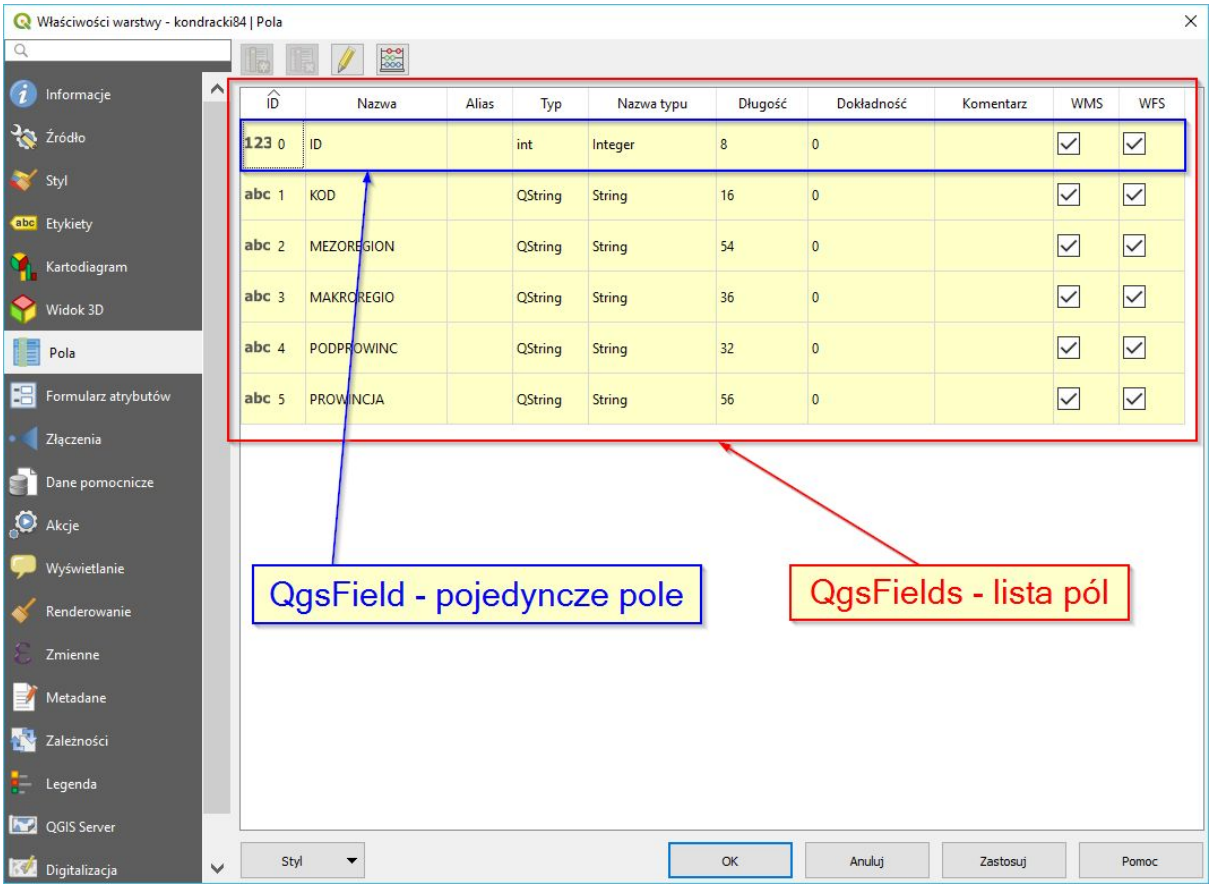
**Kod źródłowy**

```
# Pobranie aktywnej warstwy
warstwa = iface.activeLayer()
#Iteracja po obiektach przestrzennych warstwy
for obiekt in warstwa.getFeatures():
```

```
    # Pobranie geometrii obiektu
    geometria = obiekt.geometry()
    # Wyciągnięcie wierzchołków geometrii i ich wydrukowanie
    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:

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

```
# Lista nazw kolumn
tabela.names()
# [ '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 kolumny do tabeli:

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

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

# Dodanie kolumny do tabeli
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:

```
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:

```
# Stworzenie obiektu przestrzennego
obiekt = QgsFeature( tabela )
# Stworzenie geometrii
geometria = QgsGeometry.fromPointXY( QgsPointXY(20, 51) )
# Przypisanie geometrii do obiektu
obiekt.setGeometry( geometria )
# Przypisanie pojedynczej wartości
obiekt['opis'] = 'Obiekt przestrzenny'
#Przypisanie wielu wartości jednocześnie
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 w źródle, aby je zobaczyć na wczytanej warstwie przestrzennej w QGIS należy ją odświeżyć za pomocą metody **`reload()`**:

```
warstwa.reload()
```

Do zmian należy wykorzystać odpowiednie metody tej klasy:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Ćwiczenie

**Treść zadania**
Dodaj do warstwy *kondracki* dodać nową kolumnę *area* przechowującą liczby rzeczywiste i wypełnij ją wartościami reprezentującymi powierzchnię każdego poligonu wyrażoną w $km^2$.

**Opis**
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 warstwa 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()`**.

**Kod źródłowy**

```
# Pobranie aktywnej warstwy
warstwa = iface.activeLayer()
# Sterownik warstwy wektorowej
sterownik = warstwa.dataProvider()

# Utworznie nowego pola
pole = QgsField( 'area', QVariant.Double )
# Zapisanie pola w źródle danych
sterownik.addAttributes( [pole] )
# Odświeżenie informacji w QGIS odnośnie zmian w źródle danych
warstwa.reload()
# Pobranie indeksu nowego pola
indeks = warstwa.fields().indexFromName( 'area' )

# Iteracja po obiektach przestrzennych warstwy
for obiekt in warstwa.getFeatures():
    # Pobranie geometrii
    geometria = obiekt.geometry()
    # Zapisanie wartości w źródle danych, podajemy zagnieżdżone słowniki,
    # kluczem pierwszego jest identyfikator obiektu, który będzie
modyfikowany,
    # kluczem drugie indeks pola do aktualizacji
    sterownik.changeAttributeValues ({
        obiekt.id(): { indeks : geometria.area()/1000  }
    })

# Odświeżenie informacji w QGIS odnośnie zmian w źródle danych
warstwa.reload()
```

# Analizy przestrzenne

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*.

## Analiza ciągów czasowych w NumPy

Analizy ciągów czasowych służą do śledzenia zmian zjawisk w czasie i prognozowaniu ich wartości w przyszłości. Przykładem są analizy temperatur dla stacji meteorologicznych i określenie zmian klimatycznych czy dobowe/tygodniowe/miesięczne zmiany natężenia ruchu na drogach. Posiadając dane historyczne za pomocą biblioteki *NumPy* możliwe jest

sprawdzenie trendu zmian oraz interpolacja i ekstrapolacja danych. Służą do tego modele matematyczne.

## Trend i ekstrapolacja danych

Posiadając serię danych (pary liczb reprezentujących np. zmianę wartości w czasie) możliwe jest wyliczenie funkcji obrazującej trend danego zjawiska. Funkcja ta może służyć do ekstrapolacji danych poza podany zakres. Funkcje trendu mogą mieć różną postać np. liniową, wykładniczą czy logarytmiczną oraz wielomianową. Tą ostatnią można łatwo wyznaczyć z pomocą biblioteki *NumPy*. Funkcja **`polyfit`** przyjmuje dwie listy określające współrzędne X i Y oraz stopień liczonego wielomianu i zwróci listę współczynników wielomianu dla każdej jego potęgi. Do obliczeń wykorzystywana jest metoda najmniejszych kwadratów.

```
# Import biblioteki NumPy
import numpy

# Dane osi X, np. kolejne dni pomiarowe
x_data = [1, 2, 3, 4, 5]
# Dane osi Y, wartość danego zjawiska
y_data = [10, 12, 11, 12, 13]
# Obliczenie wielomianu stopnia 1 (funkcja liniowa)
coefficients = numpy.polyfit( x_data, y_data, 1 )
print( coefficients )
# [ 0.6 9.8 ] - lista współczynników wielomianu
# funkcja ma postać y = 0.6x + 9.8
```

Po obliczeniu współczynników funkcji trendu możliwa jest ekstrapolacja danych poza pierwotny zakres np. w celu obliczenia wartości jakie mogą się pojawić w przyszłości. Służy do tego metoda **`poly1d`** przyjmująca jako argument obliczone współczynniki wielomianu i zwraca obiekt reprezentujący obliczoną funkcję. Podając nowy zakres (dane osi X) można zwrócić wartości (dane osi Y).

```
poly_function = numpy.poly1d( coefficients )
print( poly_function )
# 0.6 x + 9.8
new_x_data = [ 6, 7 ]
exp_values = poly_function( new_x_data )
print( exp_values )
# [13.4 14. ] - wartości dla kolejnych zakresów
```

# Ćwiczenie

**Treść zadania**
Stwórz w module *analysis.py* funkcję **`calculate_polynomial`**, która przyjmie 3 argumenty:

- **`x_data`** - dane dla osi X, lista liczb
- **`y_data`** - wartości danych dla osi Y, lista liczb
- **`degree`** - stopień wielomianu. liczba naturalna

Wykorzystując możliwości biblioteki *NumPy* zwróć funkcję wielomianową reprezentującą wykres trendu dla podanych wartości w listach **`x_data`** i **`y_data`** o danym stopniu.

**Opis**
Funkcja **`numpy.polyfit`** przyjmuje dwie listy określające współrzędne danych (zakres i wartości) oraz stopień wielomianu i zwraca współczynniki funkcji wielomianowej. Następnie należy je przekazać do funkcji np. **`numpy.poly1d`** aby stworzyć obiekt reprezentujący obliczoną funkcję wielomianową.

**Kod źródłowy**

```
# Import biblioteki NumPy
import numpy

def calculate_polynomial( x_data, y_data, degree ):
      # Obliczenie współczynników wielomianu
      coefficients = numpy.polyfit( x_data, y_data, degree )
      # Obiekt reprezentujący funkcję
      poly_function = numpy.poly1d( coefficients )
      return poly_function
```

# Ćwiczenie

**Treść zadania**
Stwórz w module *analysis.py* funkcję **`calculate_data`**, która przyjmie 3 argumenty:

- **`poly_function`** - obiekt reprezentujący funkcję, obiekt **`numpy.poly1d`**
- **`start`** - wartość dla początku zakresu danych, liczba naturalna
- **`end`** - wartość dla końca zakresu danych, liczba naturalna

**`start`** i **`end`** określają zakres argumentów dla funkcji w formie liczb naturalnych gdzie **`start`** ≤ **`end`**. Funkcja ma zwracać dwie listy, pierwsza zawierająca zakres argumentów od **`start`** do **`end`** zwiększających się o 1, a druga obliczone za pomocą podanej funkcji wartości dla tych argumentów.

**Opis**
Posiadając wartości początkową i końcową zakresu liczb naturalnych można wyliczyć zbiór liczb rosnących od najmniejszej do największej co 1 za pomocą funkcji **`range`**. Zwraca ona zbiór liczb całkowitych z podanego zakresu w formie iteratora, który można skonwertować

na listę za pomocą funkcji **`list()`**. Mając taką listę można ją przekazać do funkcji **`func`**, w celu obliczenia wartości dla każdego jej elementu. Wynik zwracany jest jako tablica NumPy, gdzie każdy element to wartość (y) funkcji w podanym argumencie (x) - zgodnie z kolejnością podanej listy argumentów.

**Kod źródłowy**

```
def calculate_data( poly_function, start, end ):
      # Obliczenie zakresu wg podanych wartości granicznych
      new_x = list(range(start, end+1))
      # Obliczenie wartości funkcji w podanych argumentach
      new_y = poly_function( new_x )
      # Zwrócenie obu list
      return new_x, new_y
```

## Wizualizacja danych

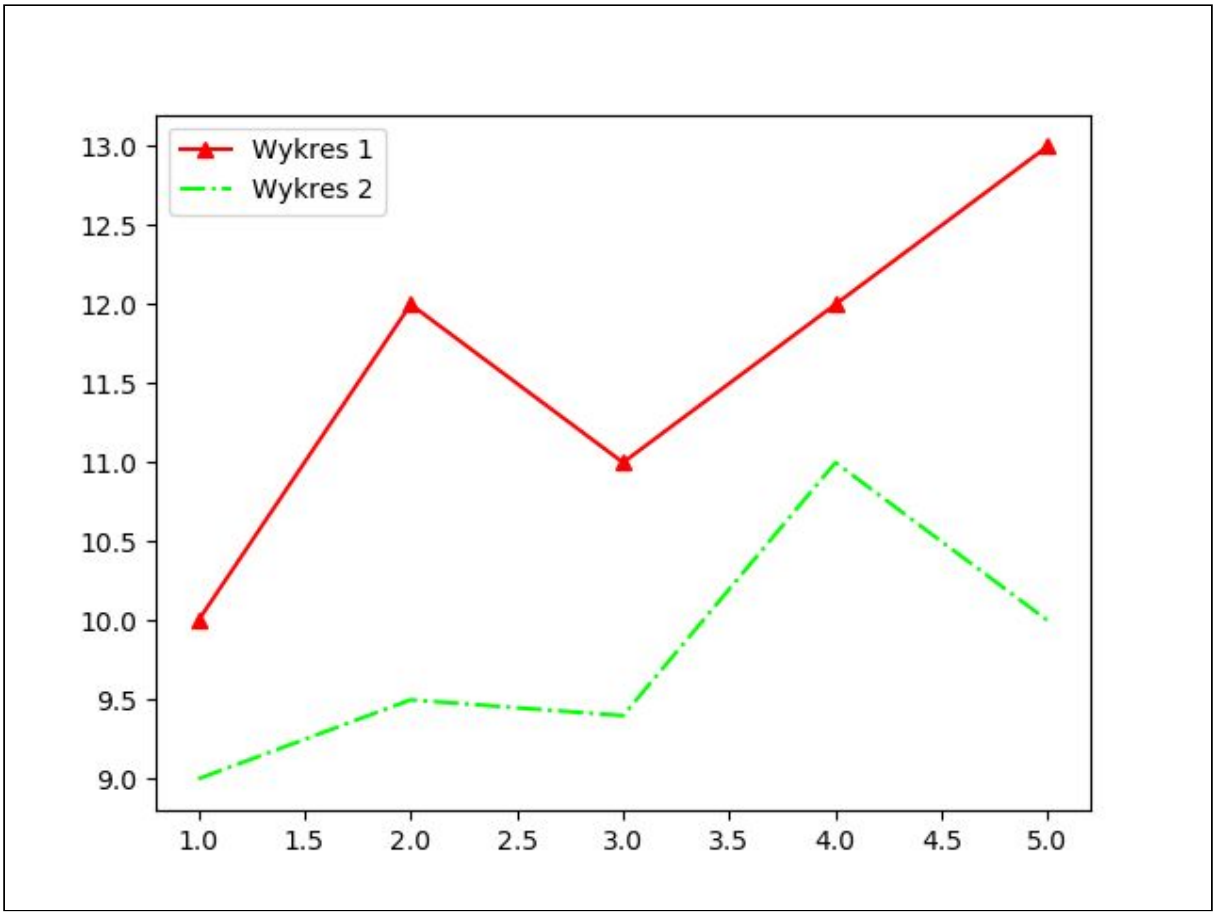
Biblioteka *matplotlib* służy do wizualizacji danych, w szczególności umożliwia tworzenie statycznych i dynamicznych wykresów. Dynamiczne wykresy są otwierane w nowym oknie, w którym możliwe jest przybliżanie i przesuwanie widoku oraz jego eksport do pliku graficznego. Aby utworzyć takie okno należy zaimportować obiekt **`pyplot`** i za pomocą metod **`plot()`** dodać dane wykresu. Funkcja ta przyjmuje wiele dodatkowych argumentów, które służą stylizacji danych m.in.:

- **`color`** - kolor linii, można podać m.in. nazwę angielską (np. **`'red'`**), zapis szesnastkowy ( **`#FF00FF`** ) lub listę wartości barw RGB( np. **`[1, 0.5, 0]`** ),
- **`linestyle`** - styl linii: solid - normalna linia, dashed - przerywana, dotted - kropkowana, dashdot - przerywana z liniami i kropkami,
- **`linewidth`** - szerokość linii,
- **`marker`** - znacznik punktu m.in.: **`'.'`** - kropka; **`'o'`** - koło; **`','`** - piskel; **`'v','<','>'`**, **`'^'`** - trójkąty; **`'s'`** - kwadrat; **`'*'`** - gwiazdka,
- **`markersize`** - rozmiar znacznika punktu,
- **`label`** - tekst w legendzie.

Wielokrotne wywołanie funkcji **`plot()`** umożliwia dodanie kilku wykresów w jednym układzie współrzędnych. Funkcja **`show()`** wyświetla przygotowany wykres.

```
# Import obiektu do tworzenia dynamicznych wykresów
import matplotlib.pyplot as plt

# Pierwszy wykres
plt.plot([1, 2, 3, 4, 5], [10, 12, 11, 12, 13],
      marker='^', color='red', label='Wykres 1' )

# Drugi wykres
plt.plot([1, 2, 3, 4, 5], [9, 9.5, 9.4, 11, 10],
      linestyle='dashdot', color=[0, 1, 0], label='Wykres 2' )
```

```
# Wygenerowanie legendy
plt.legend()
# Wyświetlenie dynamicznego wykresu
plt.show()
```




# Ćwiczenie

**Treść zadania**

Stwórz w module *analysis.py* funkcję, która wygeneruje dynamiczne wykresy dla serii danych. Jako argumenty przyjmuje ona:

- **`x_data`** - dane dla osi X, lista liczb,
- **`y_data`** - wartości danych dla osi Y, lista liczb,
- **`degree`** - stopień wielomianu. liczba naturalna, domyślnie 1,
- **`extrapolation_count`** - liczba danych, które mają być ekstrapolowane na podstawie funkcji trendu, liczba naturalna, domyślna wartość 3.

W docelowym widoku powinny znajdować się 3 wykresy:

- reprezentujący dane zdefiniowane w listach **`x_data`** i **`y_data`**, linia ciągła w kolorze *#1f77b4* z wyróżnionymi punktami danych,

- trend jako czarna linia,
- ekstrapolowane dane w formie przerywanej linii w kolorze pomarańczowym.

Do obliczeń danych potrzebnych do narysowania wykresów skorzystaj z przygotowanych wcześniej funkcji **`calculate_polynomial`** i **`calculate_data`**.

**Opis**

Na początku należy stworzyć definicję funkcji z uwzględnieniem podanych wartości domyślnych dla dwóch argumentów. Wykresy będą rysowane za pomocą metody **`pyplot.plot()`**, jako argumenty należy podać dane XY oraz parametry określające wygląd linii.
Najpierw należy narysować pierwszy wykres dla danych źródłowych czyli **`x_data`** i **`y_data`**, podając argumenty **`color='#1f77b4'`** i **`marker='o'`**.
Następnie, wykorzystując funkcję **`calculate_polynomial`** obliczamy funkcję wielomianową, która posłuży do stworzenia wykresów dla trendu i prognozowanych danych. Mając tą funkcję można skorzystać z **`calculate_data`** to wyliczenia wartości dla obu wykresów. W przypadku trendu wartość startowa i końcowa będa takie same jak w liście **`x_data`**. Zwrócone dane można przekazać do funkcji **`pyplot.plot`** dodając odpowiedni argument dla koloru czarnego.
Dla danych prognozowanych należy podać **`start`** jako o jeden większy od ostatniego elementu **`x_data`**, a **`end`** powiększony o wartość zmiennej **`extrapolation_count`**. Wynik podajemy do funkcji **`pyplot.plot`**, dodając argument dla koloru i ustawiając styl linii na przerywany ( **`linestyle='dashed'`** ).
Na koniec należy wyświetlić okno z dynamicznymi wykresami za pomocą metody **`pyplot.show()`**.

**Kod źródłowy**

```
# Import obiektu do generowania wykresów
from matplotlib import pyplot

def draw_plot( x_data, y_data, degree=1, extrapolation_count=3 ):
      # Narysowanie wykresu dla danych źródłowych
      pyplot.plot( x_data, y_data, color='#1f77b4', marker='o' )

      # Obliczenie funkcji wielomianowej
      func = calculate_polynomial( x_data, y_data, degree )

      # Wygenerowanie danych dla linii trendu
      trend_x, trend_y = calculate_data( func, x_data[0], x_data[-1] )
      # Narysowanie linii trendu na wykresie
      pyplot.plot( trend_x, trend_y, color='black' )

      # Wygenerowanie danych dla prognozy
      new_x, new_y = calculate_data( func, x_data[-1]+1,
            x_data[-1]+extrapolation_count )
```

```
    # Narysowanie linii ekstrapolacji
    pyplot.plot( new_x, new_y, color='orange', linestyle='dashed' )

    # Wyświetlenie okna z wykresem
    pyplot.show()
```

# Wtyczki QGIS

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.

W trakcie kursu zostanie przygotowana wtyczka, która na podstawie danych rastrowych wygeneruje wykresy analizujące serie danych czasowych, wraz z trendem i prognozowaniem przyszłych wartości.

## 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.

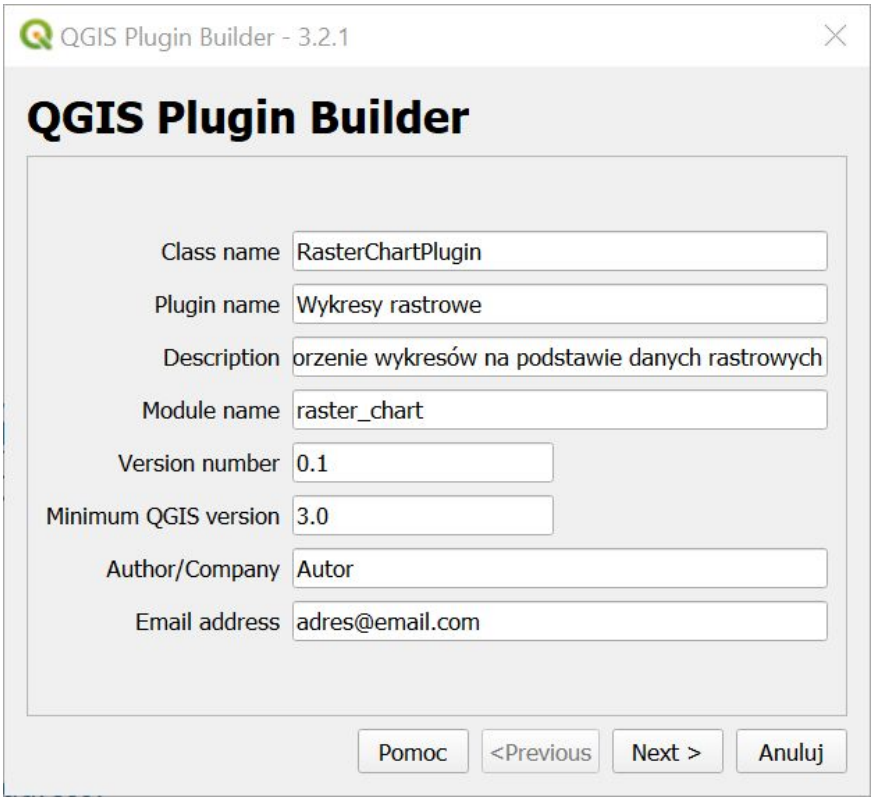
# Ćwiczenie

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

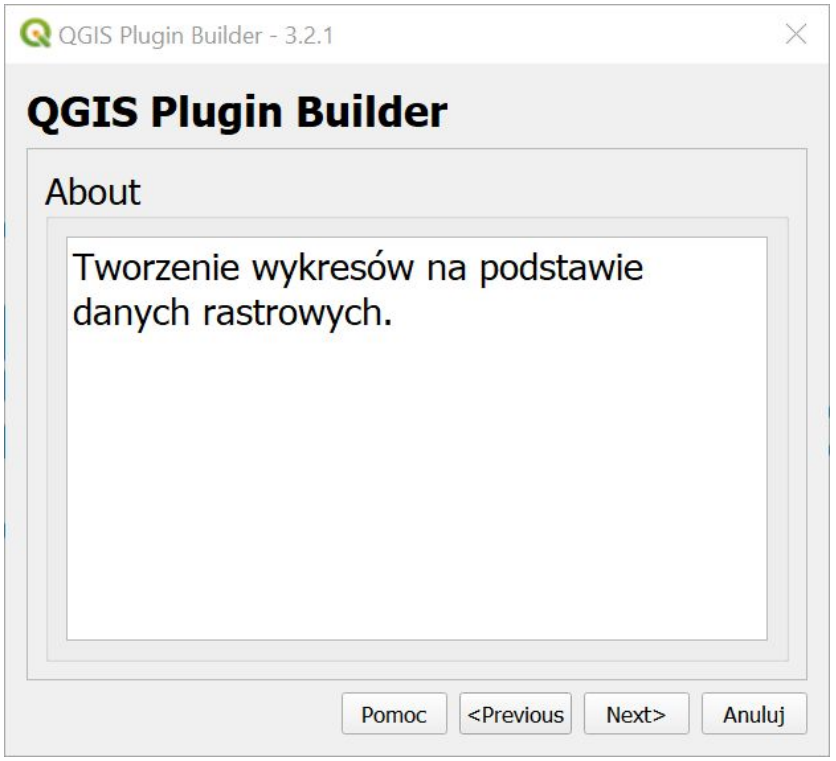
**Opis**

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

**Metadane wtyczki**

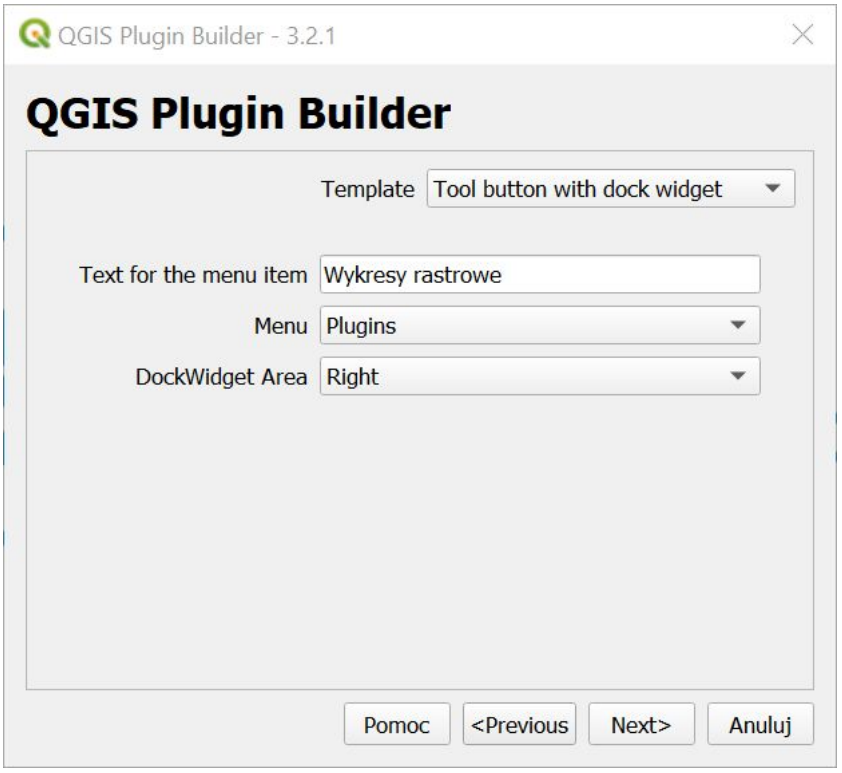



- **Class name** - nazwa klasy reprezentującej w QGIS daną wtyczkę. Nazwa nie może zawierać spacji oraz znaków specjalnych.
- **Plugin name** - Nazwa wtyczki, wyświetlana m.in. w Menedżerze wtyczek i menu.
- **Description** - krótki, jednolinijkowy opis wtyczki wyświetlany w Menedżerze wtyczek.
- **Module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

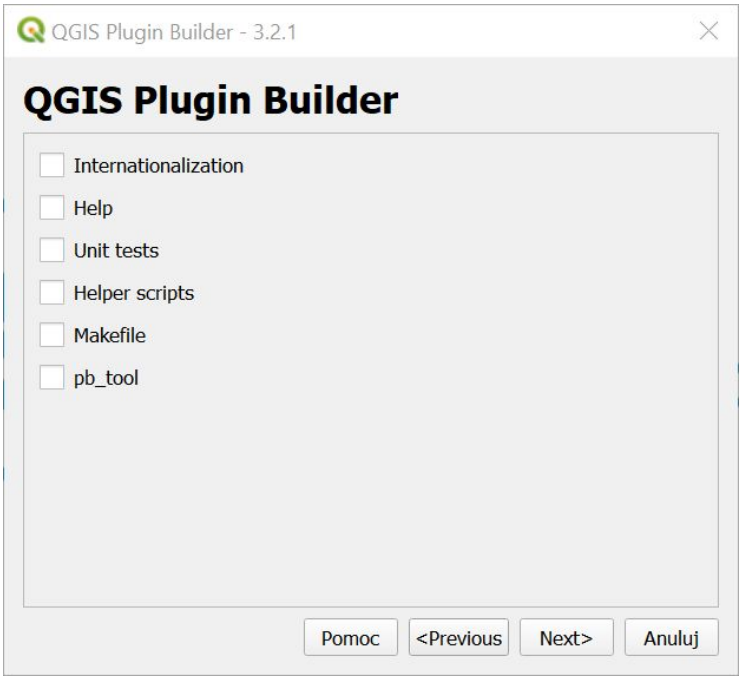
**Rodzaj okna dialogowego**




- **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.

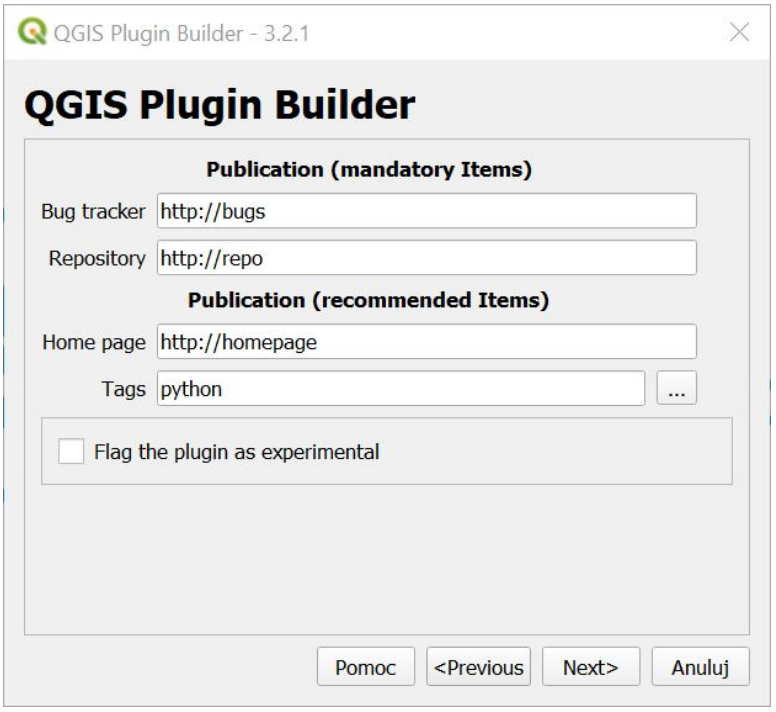
### Dodatkowe elementy




- **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.

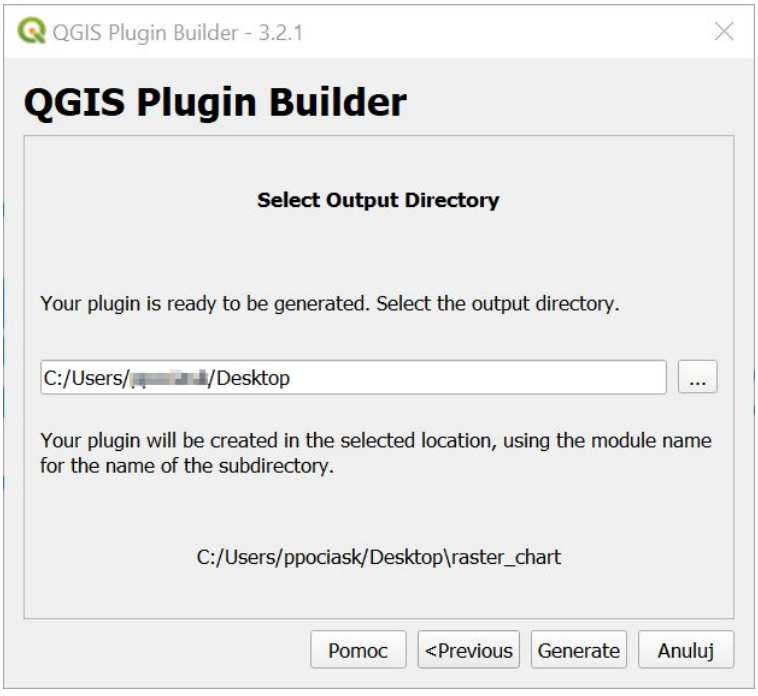
### Dodatkowe informacje

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.

### Katalog wyjściowy



## 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

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

**Opis**
Katalog wtyczek można znaleźć wybierając w menu QGIS *Ustawienia -> Profile użytkownika -> Otwórz katalog aktywnego profilu* i przechodząc do katalogu *python/plugins*. Jeśli katalog *python* nie istnieje to można go utworzyć ręcznie. Następnie kopiujemy wtyczkę do tego folderu.
W celu kompilacji wtyczki należy uruchomić konsolę *OSGeo4W Shell* i wywołać po kolei dwa polecenia:

- py3_env - ustawienie Python3 jako aktywnego
- qt5_env - ustawienie narzędzie Qt 5 jako aktywnych

Następnie wpisujemy polecenie:
`pyrcc5 -o C:/.../resources.py C:/.../resources.qrc`
Należy podać pełne ścieżki do plików, można sobie ułatwić zadanie poprzez przeciąganie plików na okno konsoli, dzięki czemy pełne ścieżki zostaną wpisane w miejscu kursora.
Po tym możliwe jest uruchomienie wtyczki, w tym celu należy zresetować QGIS (jeśli jest włączony), wejść w menu *Wtyczki -> Zarządzanie wtyczkami…*, zlokalizować wtyczkę *Wykresy rastrowe* 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

**Treść zadania**
Skonfiguruj Plugin Reloader aby możliwe było resetowanie wtyczki *Wykresy rastrowe*.

**Opis**

Należy kliknąć strzałkę znajdującą się z prawej strony ikony Plugin Reloader  i wybrać *Ustawienia*. W otwartym oknie wskazujemy wtyczkę *raster_chart* i klikamy *OK*. Po tym należy kliknąć przycisk  i sprawdzić czy wtyczka została poprawnie zresetowana.

# Framework Qt

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

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

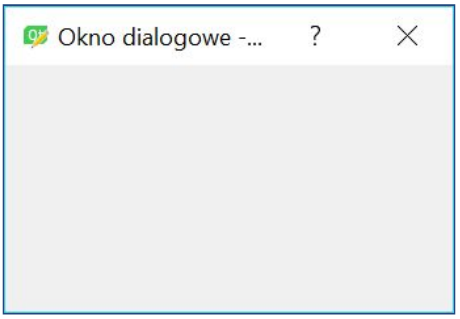
Import z głównych modułów:

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

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

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

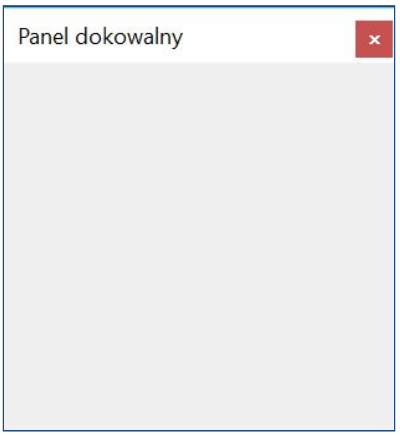
## Widżety

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

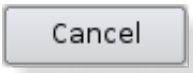
- **`QDialog`** – okno dialogowe, swobodne okno, które użytkownik może przesuwać, może ono być zawsze na wierzchu okna aplikacji oraz je zablokować (tryb modalny).



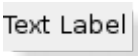

- **`QDockWidget`** - panel, który może zostać “przyklejony” do jednej z krawędzi okna głównego, panele można również dokować jeden na drugim.




- **`QPushButton`** – przycisk

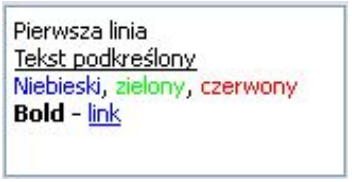



- **`QLabel`** – etykieta tekstowa

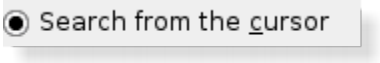



- **`QLineEdit`** – okienko do wpisywania tekstu (jednoliniowe)

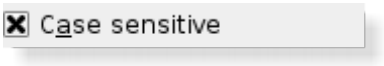

- **`QTextEdit`** – okienko do wpisywania tekstu (wieloliniowe)




- **`QRadioButton`** – przycisk opcji

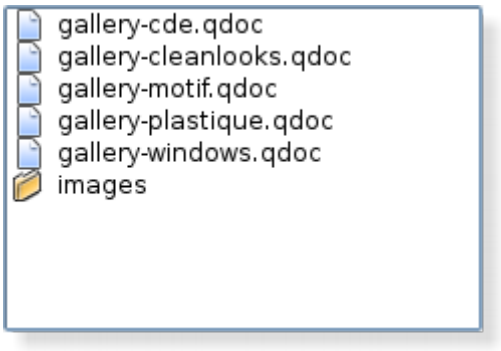



- **`QCheckBox`** – przycisk wyboru

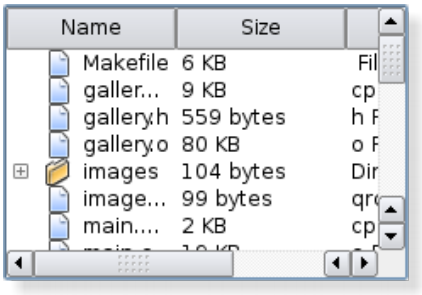



- **`QComboBox`** – lista wyboru

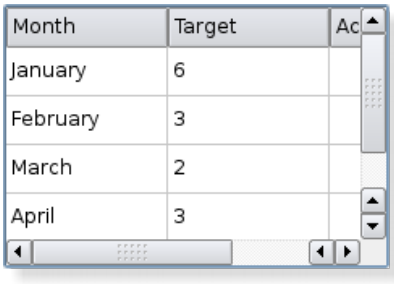

- **`QListWidget`** – lista elementów
- **`QListWidgetItem`** – element listy




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




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

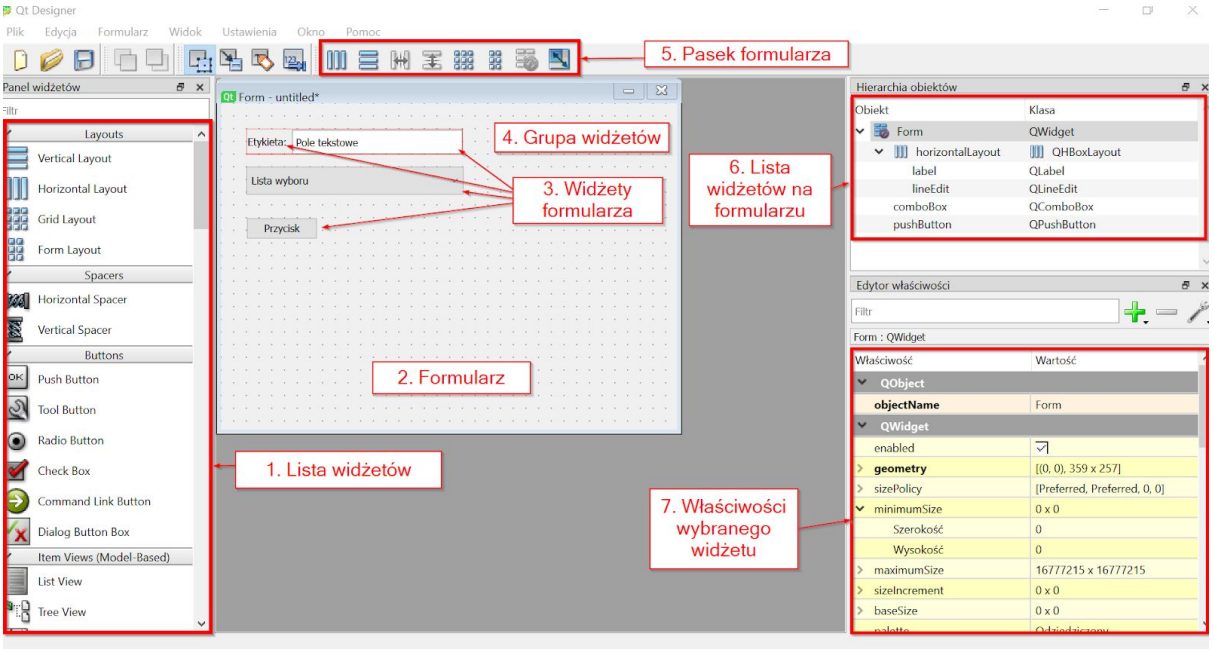



*Qt Designer* umożliwia również rejestrowanie dodatkowych widżetów. W ten sposób można również dodać wiele kontrolek pochodzących z QGIS.

## 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 do zmiany szerokości,

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

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

## Pliki .ui i Python

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

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

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

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

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

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' ))
```

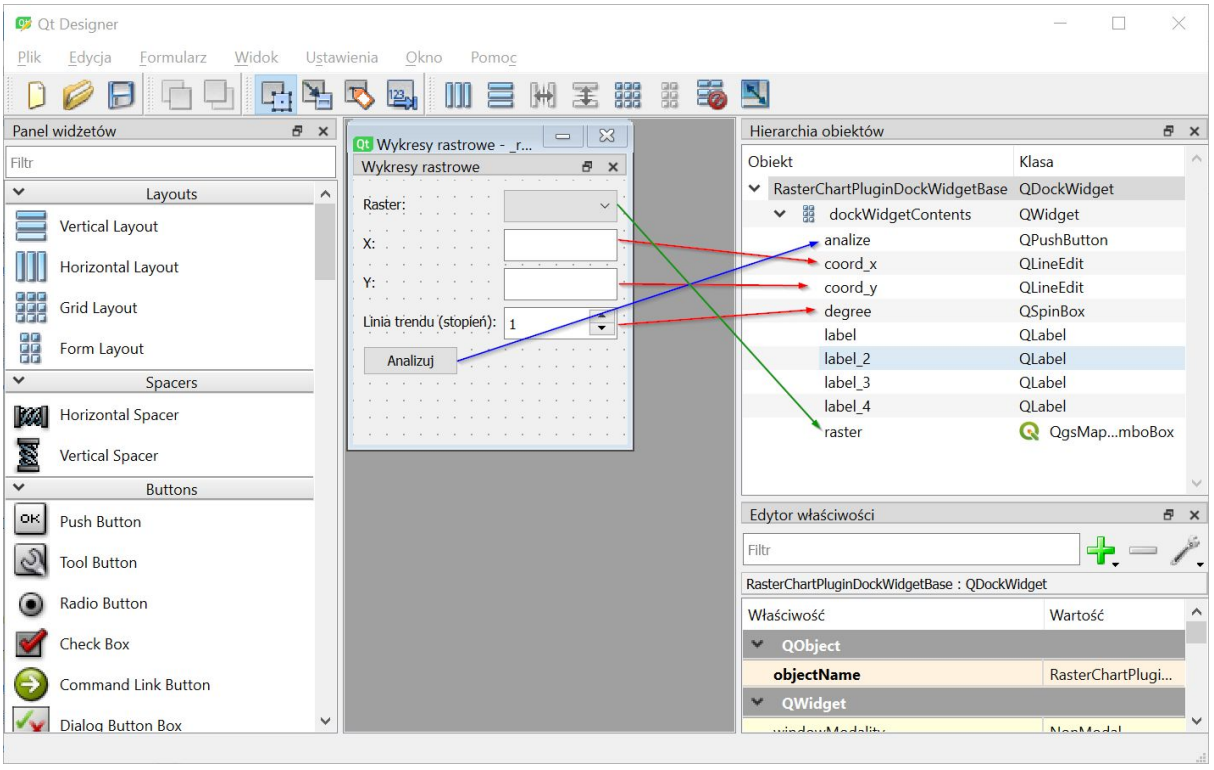
```
# Użycie stworzonego obiektu jako klasy bazowej
class PluginDockWidget(QDockWidget, FORM_CLASS):
    ...
```

Ten sposób jest wykorzystywany przez *Plugin Builder*.

# Ćwiczenie

**Treść zadania**

Za pomocą Qt Designer stwórz okno wtyczki wg poniższego wzoru.




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

**Opis**

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)
```

# Rozwijanie wtyczki

## Akcje

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

```
def add_action(
      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
)
```

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

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

# Ćwiczenie

**Treść zadania**
Dodaj nową akcję wykorzystując metodę **`add_action`**. Do akcji powinna zostać dodana nowa ikona *point_tool.png* oraz opis *Wybierz punkt na mapie*. Jako funkcję wywoływaną przez tą akcję utwórz w głównej klasie wtyczki metodę **`activatePointTool`**. Metoda ta powinna drukować tekst *Kliknięto akcję*. Akcja powinna mieć możliwość pozostawania w formie wciśniętej.

**Opis**
Na początek należy skopiować do katalogu wtyczki nową ikonę *point_tool.png*, dopisać ją w zasobach oraz skompilować.
Kolejnym krokiem jest zmodyfikowanie głównej klasy wtyczki. Nowe akcje należy definiować w metodzie **`initGui`**. Znajduje się w niej już kod dodający pierwszą akcję odpowiedzialną za wyświetlanie panelu dokowanego. Podobnie należy dodać nowy przycisk, podając dane zgodnie z opisem. W związku z tym, że przycisk ma pozostawać wciśnięty trzeba go będzie dodatkowo zmodyfikować. W tym celu wynik działania metody **`add_action`** należy przypisać do zmiennej i ustawić jej metodę **`setCheckable`** na prawdę logiczną. Na koniec należy dodać nową metodę w klasie wtyczki, która drukuje określony tekst.

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

**Kod źródłowy**

```
# Nowe akcje należy definiować w metodzie initGui
def initGui(self):
    ...
    # Stworzenie nowej akcji
    action = self.add_action(
        ':/plugins/raster_chart/point_tool.png',
        'Wybierz punkt na mapie',
        self.activatePointTool,
        parent=self.iface.mainWindow() )
    # Ustawienie możliwości pozostawienia wciśniętego przycisku
    action.setCheckable( True )

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

# Narzędzia mapy

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

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

Jednocześnie może być aktywne tylko jedno narzędzie mapy.

## QgsMapToolEmitPoint

Jest to jedno z najprostszych narzędzi mapy. Pozwala na zwrócenie punktu (instancja klasy **`QgsPonitXY`**), w którym użytkownik kliknął na mapę oraz przycisku myszy, który był użyty. Służy do tego sygnał **`canvasClicked`**, do którego można się podpiąć. Współrzędne zwracane są w jednostkach układu współrzędnych projektu QGIS.

Tworząc instancję tej klasy należy podać jako argument klasę **QgsMapCanvas** reprezentującą okno mapy, na którym ma być ono używane. W tym celu najprościej wykorzystać obiekt iface.

```
# Stworzenie nowego narzędzia mapy
narzedzie = QgsMapToolEmitPoint( iface.mapCanvas() )
```

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

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

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

## Aktywacja narzędzi mapy

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

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

## Akcje i narzędzia mapy

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

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

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

# Ćwiczenie

**Treść zadania**

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

**Opis**

W pierwszej kolejności należy utworzyć nową instancję klasy **`QgsMapToolEmitPoint`** w metodzie **`initGui`**. Obiekt ten będzie wywoływany w metodzie **`activatePointTool`**, dlatego należy do zapamiętać jako atrybut klasy. Klasę **`QgsMapToolEmitPoint`** należy zaimportować z modułu **`qgis.gui`**. Następnie przypisujemy do niego wcześniej utworzoną akcję za pomocą metody **`setAction`**.

Kolejnym krokiem jest aktywacja narzędzia, odbywa się to w metodzie **`activatePointTool`**, która jest powiązana z przypisaną akcją. Dodajemy w niej aktywację narzędzia z poziomu klasy **`QgsMapCanvas`** za pomocą metody **`setMapTool`**. To pozwoli włączać i wyłączać narzędzie. Działanie należy zweryfikować w QGIS.

Ostatnim etapem jest obsługa kliknięcia użytkownika na mapie w momencie gdy narzędzie jest aktywne. W tym celu należy powiązać jego sygnał **`canvasClicked`** z metodą, która ma być dodana w klasie panelu. Takie powiązanie może odbyć się dopiero po tym jak jest utworzona instancja tej klasy. W tym celu należy znaleźć miejsce w kodzie głównej klasy wtyczki gdzie to się odbywa tj. w metodzie **`run`**. Pod linijką z tworzeniem instancji należy dopisać powiązanie sygnału z metodą **`pointClicked`**.

Ostatnim etapem jest dodanie metody **`pointClicked`** do klasy **`RasterChartPluginDockWidget`**. Można w niej wydrukować przekazywany punkt, w celu zweryfikowania czy narzędzie działa poprawnie.

**Kod źródłowy**

```
# plik raster_chart.py

#Import
from qgis.gui import QgsMapToolEmitPoint

def initGui(self):
    ...
    # Utworzenie nowego narzędzia mapy
    self.pointTool = QgsMapToolEmitPoint(self.iface.mapCanvas() )
    self.pointTool.setAction( pointAction )

def activatePointTool(self):
    # Aktywacja narzędzia
```

```
    self.iface.mapCanvas().setMapTool( self.pointTool )

def run(self):
    ...
    if self.dockwidget is not None:
        ...
        # Powiązanie kliknięcia na mapie z funkcją
        self.pointTool.canvasClicked.connect(
            self.dockwidget.pointClicked )

# plik raster_chart_dockwidget.py

def pointClicked(self, point, button):
    print( point )
```

## Dostęp do widżetów

Okna dialogowe są zdefiniowane jako klasy. Przy ich tworzeniu wykorzystywane są inne klasy, które zostały wygenerowane z plików .ui, w których są tworzone i układane na formularzu widżety. Po wywołaniu metody **`setupUi`** z poziomu tej klasy mamy dostęp do wszystkich widżetów, które zostały utworzone w *Qt Designer*. Są one zdefiniowane jako atrybuty klasy więc wystarczy podać ich nazwę (atrybut **`objectName`** w *Qt Designer*) po obiekcie **`self`**:

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

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

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

Wszystkie metody danego typu widżetu są opisane w jego dokumentacji.

### QLineEdit

Prosta kontrolka do wprowadzania tekstu w jednej linii.

- **`setText( str ), text()`** - ustawienie i zwrócenie wyświetlanego tekstu.
- **`setPlaceholderText( str ), placeholderText()`** - ustawienie i zwrócenie tekstu, który jest wyświetlany jeśli pole tekstowe jest puste. Tekst taki jest lekko wyszarzony.
- **`setMaxLength( int ), maxLength()`** - ustawienie i zwrócenie maksymalnej długości tekstu, jaki można wprowadzić. Dłuższe teksty są skracane.
- **`setReadOnly( bool ), isReadOnly()`** - ustawienie i zwrócenie informacji o tym, czy jest możliwa edycja wartości przez użytkownika.

### QSpinBox i QDoubleSpinBox

Kontrolki pozwalające na wprowadzanie wartości numerycznych. **`QSpinBox`** pozwala wyświetlać liczby całkowite (**`int`**), a **`QDoubleSpinBox`** rzeczywiste (**`float`**).

- **`setValue( int/float ), value()`** - ustawienie i zwrócenie wyświetlanej wartości liczbowej.
- **`setMinimum( int/float ), setMaximum( int/float ), minimum(), maximum()`** - metody ustawiające i zwracające minimalne i maksymalne wartości możliwe do wprowadzenia.
- **`setDecimals( int ), decimals()`** - dotyczy **`QDoubleSpinBox`**, ustawienie i zwrócenie dokładności liczb rzeczywistych (wyświetlanych miejsc po przecinku).

### QgsMapLayerComboBox

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

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

# Ćwiczenie

**Treść zadania**

Dopisz w metodzie **`pointClicked`** wstawienie wartości współrzędnych punktu do widżetów edycyjnych w panelu wtyczki.

**Opis**

Metoda pointClicked otrzymuje dwa argumentu, pierwszy z nich to punkt (**`QgsPointXY`**). Współrzędne są dostępne za pomocą metod **`x`** i **`y`**. Zwracają one wartości liczbowe. Aby je wyświetlić w polach edycyjnych należy dokonać konwersji na tekst za pomocą funkcji **`str`**. Klasa **`QLineEdit`** posiada metodę **`setText`**, dzięki której można ustawić w niej wyświetlany tekst. Należy wywołać tą metodę dla obu pól edycyjnych przekazując jako argumenty współrzędne punktu.

**Kod źródłowy**

```
def pointClicked(self, point, button):
    # Wyciągnięcie współrzędnych punktu i konwersja do tekstu
    x = str( point.x() )
    y = str( point.y() )
    # Ustawienie tekstu dla kontrole edycyjnych QLineEdit
    self.coord_x.setText( x )
    self.coord_y.setText( y )
```

# Ćwiczenie

**Treść zadania**

Wykorzystując dokumentację QGIS API ustaw dla kontrolki **`QgsMapLayerComboBox`** filtr, dzięki któremu wyświetlane będą tylko warstwy rastrowe. Następnie dodaj metodę **`startAnalysis`**, która będzie uruchomiona w momencie kliknięcia przycisku *Analizuj*. Wewnątrz metody zbierz do zmiennych parametry ustawione przez użytkownika:

- **`raster`** - **`QgsRasterLayer`**, wybrana warstwa rastrowa,
- **`point`** - **`QgsPointXY`**, punkt kliknięcia,
- **`degree`** - **`int`**, stopień linii trendu

**Opis**

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

W celu obsługi przycisku *Anuluj* należy podpiąć się do jego sygnału **`clicked`**. Jest on wysyłany, gdy użytkownik kliknie w przycisk. Następnie należy zdefiniować slot, czyli funkcję, która ma być wywołana w momencie wysłania sygnału. W jej wnętrzu definiowane są zmienne przechowujące informacje od użytkownika:

- **`raster`** - warstwa rastrowa definiowana jest w widżecie **`QgsMapLayerComboBox`**, który posiada metodę currentLayer. Zwracana ona warstwę wczytaną w QGIS, która jest wybrana w polu wyboru jako klasę **`QgsRasterLayer`**.
- **`degree`** - stopień wielomianu jest ustawiany w kontrolce **`QSpinBox`**. Posiada ona metodę value do zwracania wyświetlanej liczby.
- **`point`** - współrzędne punktu są przechowywane w dwóch polach edycyjnych (**`QLineEdit`**). Do pobrania ich zawartości służy metoda **`text`**. Zwraca ona łańcuch

znaków (**`str`**), który należy skonwertować do liczby rzeczywistej (**`float`**). Mając obie liczby można utworzyć instancję klasy **`QgsPointXY`**.

**Kod źródłowy**

```
# Import klasy z QGIS API
from qgis.core import QgsMapLayerProxyModel, QgsPointXY

def __init__(self, parent=None):
      ...
      # Ustawienie filtra dla listy warstw
      self.raster.setFilters( QgsMapLayerProxyModel.RasterLayer )
      # Połączenie przycisku Analizuj z metodą klasy
      self.analize.clicked.connect( self.startAnalysis )

def startAnalysis(self):
      # Wybrana warstwa rastrowa (QgsRasterLayer)
      raster = self.raster.currentLayer()
      # Stopień wielomianu (liczba naturalna)
      degree = self.degree.value()
      # Wyciągnięcie współrzędnych z pól edycyjnych i konwersja na liczby
      x = float( self.leX.text() )
      y = float( self.leY.text() )
      # Utworzenie punktu
      point = QgsPointXY( x, y )
```

# Ćwiczenie

**Treść zadania**

Dodaj stworzony we wcześniejszych ćwiczeniach moduł *analysis.py* do wtyczki. Wykorzystując zebrane parametry wykonaj identyfikację wybranego rastra we wskazanym punkcie i wyświetl wykres obrazujący dane z rastra, linię trendu i ekstrapolację danych. Analiza ma zostać wykonana po kliknięciu przez użytkownika przycisku *Analizuj*.

**Opis**

Aby dodać moduł do wtyczki należy skopiować plik *analysis.py* do jej katalogu. Następnie należy w module panelu dokowanego zaimportować dwie funkcje z dodanego pliku, które będą potrzebne do wykonania analizy: raster_identify i draw_plot..
W pierwszej kolejności do funkcji startAnalysis należy dopisać identyfikację rastra w punkcie. Służy do tego funkcja raster_identify, która wymaga podania dwóch argumentów:

- raster - QgsRasterLayer, warstwa do identyfikacji
- point - QgsPointXY, miejsce analizy

Oba argumenty są podawane przez użytkownika i są zapisane w odpowiednich zmiennych. Funkcja zwraca dwie listy określające współrzędne X (lata) i Y (wartości) zebranych danych. Posłużą one do wywołania drugie funkcji z modułu analitycznego tj. draw_plot. Dodatkowo

należy podać trzeci argument określający stopień wielomianu używanego do obliczenia linii trendu i ekstrapolacji danych.

**Kod źródłowy**

```
# Import funkcji z modułu analitycznego
from .analysis import raster_identify, draw_plot

# Funkcja wywoływana przez przycisk Analizuj
def startAnalysis(self):
    ...
    # Identyfikacja rastra
    dane_x, dane_y = raster_identify( raster, point )
    # Wygenerowanie wykresu
    draw_plot( dane_x, dane_y, degree )
```