Programowanie geoinformatyczne w języku Python

Jarosław Jasiewicz

14 października 2020

Wprowadzenie

Tu będzie wstęp w tym dokumencie

Rozdział 1

Modele danych geoprzestrzennych

1.1 Czym jest Warstwa Abstrakcji Danych Geograficznych

Dane geograficzne, zarówno siatki (dane rastrowe) jak i dane wektorowe mogą być zorganizowane w bardzo różny sposób. Zróżnicowanie wewnętrznej organizacji danych ma swój początek w licznych misjach pomiarowych, których wyniki były udostępniane różnych formatach. Oznacza to że każda misja stosowała inną kolejność bitów (big endian i little endian), inny sposób definiowania początku siatki (*origin*) inny sposób definiowania komórki (narożnik lub środek) czy wreszcie inny sposób wewnętrznej organizacji pliku. Znakomita większość tych różnic miała charakter czysto umowny - ktoś po prostu podjął decyzję, że dane będą zorganizowane w taki a nie inny sposób. W praktyce jednak, dowolnie sformatowane dane geoprzestrzenne mogą być sprowadzone do jednolitego formatu - oczywiście osobno dla danych wektorowych jak i rastrowych - ten jednolity format to właśnie Warstwa Abstrakcji Danych Geograficznych. Oryginalnym pomysłodawcą tego rozwiązania był Frank Warmerdam, obecnie projektem opiekuje się FUndacja OSGeo: akronim Open Source Geospatal. GDAL oryginalnie obsługiwał wyłącznie formaty rastrowe, niemniej jednak od ponad 10 lat rozwijany jest wspólnie z wcześniej niezależnymi projektami **OGR** (Open Geospatial Resources), **OSR** (Open Spatial Reference) i **GNM** (Geographic Network Model). Wszystkie te projekty rozwijane są pod wspólnym parasolem **GDAL**. Rozprowadzane są jako jedna biblioteka składająca się - w zależności od języka - z kilku modułów. Biblioteka jest rozprowadzona na licencji X/MIT i jest wykorzystywana praktycznie we wszystkich otwarto-źródłowych i większości komercyjnych projektów związanych z systemami informacji geograficznej.

Wszystkie moduły wchodzące w skład biblioteki GDAL zostały napisane w językach C++ i C, ale fundacja OSGeo oficjalnie wspiera dowiązania (*ang. bindings*) do języków C, C++, Java i najbardziej nas interesujący - Python. Poza tym rozwijana są dowiązania do innych języków - takich jak Rust, Julia, Go. W przypadku jeżyka R istnieją narzędzia - zarówno starszy rGDAL jak i nowsze: sf, raster i stars, które pozwalają na importowanie danych geoprzestrzennych za pośrednictwem GDAL, nie są to jednak klasyczne dowiązania - a więc wywoływania funkcji z poziomu języka - a rozbudowane systemy zarządzania danymi geoprzestrzennymi dostosowane do architektury języka R.

1.1.1 GDAL jako dowiązania do biblioteki C

Dowiązania to jest jest część API (interface programowania aplikacji), która pozwala zbudować połączenia (dosłownie kod-klej) pomiędzy aktualnie używanym językiem programowania a skompilowaną biblioteką innego języka. Ze względu na wydajność większość procedur obliczeniowych jest tworzona w językach C lub C++ lub Fortran (starszy kod), języki programowania wysokiego poziomu - o złożonej składni i architekturze korzystają właśnie z dowiązań pozwalających na bezpośrednie wywoływanie procedur napisanych w wysoko wydajny sposób z poziomu języka, który wydajność poświęcił na rzecz prostoty pisania kodu.

Dowiązania powodują, że raz dobrze napisany kod może być stosowany w innych językach o ile oferują one mechanizm dowiązań. Dowiązania stosuje się również w celu przyspieszania algorytmów. Jeżeli jakaś procedura jest kluczowa dla wykonywania obliczeń, można ją napisać w szybszym języku (z reguły C) a następnie połączyć ze znacznie łatwiejszym w zarządzaniu kodem Pythona. GDAL/OGR korzystają z mechanizmu SWIG (akronim: *Simplified Warper and Interface Generator*). Mechanizm ten opiera się na tworzeniu uniwersalnych plików interface, zawierających listę udostępnionych funkcji języka C/C++ wraz z ich nagłówkami. Następnie mechanizm SWIG kompiluje kod C/C++ do postaci binarnej oraz funkcje natywne dla danego języka. Zaletę mechanizmu SWIG i innych jemu podobnych, jak na przykład SIP, jest jego uniwersalność, raz napisany plik interface może być użyty w każdym z języków obsługujących ten mechanizm.

Dowiązania mają swoją wadę: przenoszą mechanizmy i sposób myślenia jednego języka na drugi. O ile dowiązania dotyczą pojedynczych funkcji - taka wada nie jest zauważalna. Jeżeli dotyczą całego systemu bibliotek - staje się to poważnym problemem, zwłaszcza dla osób, które nie znają języka C/C++. Oryginalna biblioteka GDAL nie jest wygodna w użyciu, z tego powodu w języku Python rozwijane są narzędzia, które, podobnie jak w języku R mają ukryć złożoność bibliotek języka C++ i udostępniać dostęp do danych geoprzetrzennych w sposób naturalny dla użytkowników języka Python. Biblioteki te zostaną omówione w osobnym rozdziale.

W bibliotece GDAL, przykładem takiego przenoszenia mechanizmów innego języka jest dostęp do poszczególnych elementów poprzez indeks – a więc w sposób swobodny typowy dla C/C++ czy Javy a nie w sposób sekwencyjny – poprzez iteracyjne przemieszczanie się po kolejnych składowych obiektu – czyli w sposób właściwy dla Pythona. Porównajmy:

```
for element in object:
        ...

for i in range(nelements):
        element[i]
```

Dostęp swobodny i sekwencyjny Random access sequential accsess

Aby zainstalować gdal w wirtualnym środowisku Pythona na systemach linux lub Mac przy pomocy ‘pip‘, należy upewnić się czy:

1. Jest zainstalowana wersja developerska libgdal (-dev lub -devel, w zależności od dystrybucji)
2. sprawdzić wersję binarnej biblioteki gdal przy pomocy polecenia: `gdal-config --version`
3. Zainstalować wersję biblioteki pythona zgodną z zainstalowaną wersją binarną: `pip3 install gdal==X.x.x‘, gdzie X.x.x` wersja gdal zwrócona przez `gdal-config`

W przypadku systemu Windows i środowiska Anakonda pakiet GDAL jest już skompilowany pod daną wersję biblioteki.

1.1.2 Sterowniki

Abstrakcyjny model danych zarówno dla danych rastrowych jak i wektorowych jest wewnętrznie bardzo złożony i wiele rozwiązań może się wydawać błędnych albo nadmiarowych. W efekcie model ten nie jest z programistycznego punktu widzenia spójny. Z punktu widzenia Systemów Informacji Geograficznej dane są zorganizowane w postaci warstw, a te mogą być zbiorem rastrowym, poligonowym, liniowym, punktowym lub też kolekcją geometrii. Z tym, że cztery ostatnie określa się bardziej ogólnym pojęciem, jako dane wektorowe. Jest to sposób, w jaki dane postrzega użytkownik SIG. To rozróżnienie wynika z faktu, że zasadniczo istnieją osobne algorytmy dla danych rastrowych i wektorowych, a nawet w grupie danych wektorowych zasadniczo używa się innych narzędzi dla poszczególnych typów danych. Na przykład pomiar powierzchni ma sens tylko dla poligonów a długości tylko dla linii. Operację próbkowania rastra można wykonać tylko dla punktów natomiast dla poligonów podobną operacją będą statystyki zonalne.

Sam sterownik to po prostu mechanizm dostępu do źródła danych geoprzestrzennych (*Data Source*)- czyli obiektu systemu plików lub relacyjnej bazy danych. Formaty zapisu danych geoprzestrzennych są bardzo zróżnicowane różne formaty obsługują bardzo różne funkcjonalności, w związku z tym formaty abstrakcyjne muszą obsługiwać je wszystkie, nawet jeżeli występują one w nielicznych rozwiązaniach. Na przykład abstrakcyjny model danych wektorowych obsługuje możliwość tworzenia przypisywania wielu geometrii do jednego obiektu, ale większość formatów zapisu danych wektorowych (w tym format ESRI Shapefile) takich możliwości nie oferują. Z tego powodu pracując z określonymi typami danych należy zapoznać się z ograniczeniami (najczęściej sekcje *limitations* lub *issues* w dokumentacji sterowników do poszczególnych formatów).

1.2 Model danych rastrowych

Model danych rastrowych nie jest zdefiniowany w formie jednolitego standardu, gdyż większość formatów danych nie jest w stanie obsługiwać wszystkich dostępnych opcji. Model danych rastrowych opisany w oryginalnym podręczniku GDAL: https://gdal.org/user/raster_data_model.html jest luźno oparty na OpenGIS Grid Coverages specification (nie jest to standard Open Geospatial Consorcium).

Model danych zakłada że zbiór danych (*dataset*) to połączenie jednej lub więcej warstw

rastrowych oraz informacji odnoszących się do całości zbioru. Każda warstwa rastrowa (*raster band*) posiada dodatkowo swój własny zbiór informacji, odnoszący się wyłącznie do tej warstwy. Dodatkowo, zarówno cały zbiór danych jak i poszczególne warstwy mogą posiadać metadane w postaci par: nazwa - wartość. Opis zawarty na stronach podręcznika definiuje wszystkie możliwe informacje zawarte w zbiorze danych rastrowych, jednakże większość z nich ma charakter wysoce specjalistyczny - odnosi się do bardzo wąsko definiowanych źródeł danych i nie będzie tu omawiana.

Najważniejsze składowe pliku geoprzestrzennego to:

Układ odniesienia Zdefiniowany jako łańcuch Well Known Text (WKT) WKT szczegółowo zostanie omówiony [tutaj]

Geotransformaty Współczynniki transformacji afinicznej macierzy danych wg formuły:

$$X_{geo} = GT_0 + Xpixel * GT_1 + Yline * GT_2$$

$$Y_{geo} = GT_3 + Xpixel * GT_4 + Yline * GT_5$$

Gdzie: GT_0 i GT_3 odpowiednio współrzędne X i Y lewego górnego narożnika macierzy danych; GT_1 i GT_5 odpowiednio rozpiętość komórki siatki w poziomie i w pionie, GT_5 jest z reguły ujemna; dla danych zorientowanych w kierunku północnym GT_2 i GT GT_4 wynoszą 0. Wartości różne od 0 są stosowane jedynie w przypadku zbiorów danych o innej orientacji i oznaczają odpowiednio przesunięcie wiersza lub komórki w wierszu o pewną wartość.

Metadane Zawierają dane pomocnicze będące parą nazwa: wartość. Najważniejsze z nich to area or point i nodata. Pozostałe metadane są specyficzne dla źródeł danych np. dla zobrazowań satelitarnych, podzbiorów danych, specyficznych informacji na temat geometrii obrazu itp. Zrozumienie większości z nich wymaga specjalistycznej wiedzy domenowej i nie będzie tu omawiana. Więcej informacji można znaleźć w podręczniku GDAL oraz odpowiednich RFC:

1. **AREA_OR_POINT**: Informacja czy wartość komórki to wartość środka czy wartość reprezentatywna dla całego obszaru komórki
2. **NODTA_VALUES**: lista wartości reprezentujących wartości puste w poszczególnych warstwach

GCP - Ground control points opcjonalna lista współrzędnych przypisanych do jednej lub komórek. Nie są obsługiwane przez model danych bezpośrednio, taka transformacja musi być obsłużona przez aplikację.

RFC - Request for comments. Rodzaj dokumentu tworzonego przez instytucje lub indywidualnych twórców dotyczące standardów wymiany danych w internecie. Nazwa ma na celu unikanie zbyt deklaratywnego stylu, umożliwienie dyskusji i wypracowywanie konsensusu. Stosowana nie tylko w technologiach internetowych, ale również w szeroko pojętej inżynierii tworzenia oprogramowania, czy inżynierii. RFC pozostawiają wiele kwestii otwartych do dalszego uzgodnienia lub zmian jeżeli technologia ulegnie zmianie.

1.2.1 Warstwy rastrowe

W wielu przypadkach jedna warstwa rastrowa reprezentuje cały zbiór danych. Często jednak zbiór danych składa się z wielu warstw (na przykład obraz RGB lub wielokanałowy), co więcej poszczególne warstwy mogą być różnych typów. Wymagane są jedynie typ danych rozmiar i wielkość bloku. Pozostałe informacje mają charakter opcjonalny, nie wszystkie są dostępne dla każdego formatu danych. Najważniejsze właściwości warstw to:

rozmiar szerokość i wysokość, powinny być tożsame ze zbiorem danych;

typ danych dopuszczalne typy danych to: Byte (UInt8), UInt16, Int16, UInt32, Int32, Float32 i Float64. Stosowane są również typy zespolone: CInt16, CInt32, CFloat32, and CFloat64;

wielkość bloku danych jak zapisywane są dane, użyteczne w celu przyspieszenia procesu odczytu zapisu danych;

nodata value wartość reprezentująca wartość pustą. Może być zastąpiona przez *nodata mask*: opcjonalną warstwę wskazującą, które piksele nie posiadają wartości w danej warstwie.

statystyki lista wartości zawierająca opisowe statystyki warstwy, średnia, minimum, maximum, odchylenie standardowe, udział nodata;

informacje o jednostkach informacje o rodzaju stosowanych jednostek i ich transformacjach do układu SI (np. stopy na metry);

interpretacja barwna (tylko dla typów całkowitych) jakie pasmo barwne przedstawia dana warstwa np. CGI_Red. Domyślnie niezdefiniowana;

Interpretacja barwna piramidy i podgląd (overviews): dodatkowe warstwy o niższej rozdzielczości przyspieszające wyświetlanie.

Nodata - wartość pusta. W procesie zarządzania danymi występują dane, których wartość nie jest znana. Brak informacji o wartości może mieć różne źródła: brak pomiaru, błąd pomiaru, niemożność wykonania pomiaru, bezsensowność pomiaru (np. wysokość n.p.m na morzu, głębokość oceanu na lądzie itp.). Nie oznacza wartości 0, ale wartość nieokreśloną. Liczby nie mają wartości która symbolizuje wartość pustą. Najczęściej do symbolizowania wartości pustej wybiera się jedną liczbę, której pojawienie się w zbiorze danych jest niemożliwe (np, -9999 dla wartości wysokości na lądzie) i markuje się ją w nagłówku zbioru danych jako wartość pustą. Drugim sposobem jest tworzenie osobnej warstwy - maski, która zaznacza przy pomocy wartości logicznych ‘True/False‘ czy w danym miejscu wartość występuje czy nie.
Poza wartościami pustymi, określanymi jako ‘na‘ - not avialable, występują jeszcze dwa rodzaje wartości nieokreślonych: ‘nan‘ - not a number np efekt wyciągania pierwiastka drugiego stopnia z liczby ujemnej w ciele liczb rzeczywistych, oraz ‘inf‘ - najczęściej w zaawansowanych systemach numerycznych efekt dzielenia przez 0 lub liczbę bliską zeru.

1.2.2 Wielowymiarowy model danych rastrowych

Od wersji 3.1 GDAL został wprowadzony nowy model danych - oparty o wielowymiarowe macierze danych. Model wielowymiarowy jest generalizacją tradycyjnego modelu 2D do

3D lub 4D. Jest inspirowany modelem danych HDF5. Model ten nie jest oparty na tradycyjnych warstwach (Raster Bands) ale jest możliwa konwersja pomiędzy tradycyjnym a wielowymiarowym modelem danych.

1.3 Model danych wektorowych

Podobnie jak w przypadku modelu danych rastrowych, model danych wektorowych jest luźno opaty na OpenGIS simple feature. Model ten nie jest do końca przejrzysty, posiada kilka komplikacji wynikających z konieczności dostosowania go do architektury simple feature. Z tego powodu dostęp do niektórych funkcjonalności możliwy jest z różnych poziomów (np. poziomu warstwy lub definicji obiektu (feature), stąd zarówno w opisie modelu jak i później jego implementacji pojawią się pewne nielogiczności, wymagające dodatkowego komentarza.

Definicja Feature. Jest to pojęcie trudno przetłumaczone na język polski, gdzie nie ma dobrego odpowiednika. Dosłowne tłumaczenie "cecha" nie jest komunikatywne i w praktyce stosowane. Z kolei termin "obiekt" zwłaszcza w informatyce jest zastrzeżony do innego pojęcia i może być dwuznaczny. W związku z tym w podręczniku, będę stosował nie do końca dosłowne, ale dopuszczalne tłumaczenie 'element', jako najbardziej jednoznaczny i odpowiadający charakterowi pojęcia 'feature'.

Model zakłada, że**zbiór danych** wektorowych (*ang. dataset*) to jedna lub więcej **warstw** (*ang.layers*). Zbiór danych jest pojęciem abstrakcyjnym a jego rzeczywista implementacja może się różnić, w zależności od**sterownika**. Warstwa jest również pojęciem abstrakcyjnym, którego implementacja zależy od sterownika i stanowi kolekcję**elementów** (features), które posiadają ujednoliconą strukturę zdefiniowaną w **klasie elementów** (*ang. feature definition*). Ta ostatnia jest unikalna dla warstwy ale stanowi osobną klasę obiektów dublującą funkcjonalność warstwy i stąd jest źródłem zamieszania.

Nadrzędnym elementem dla modelu danych wektorowych jest sterownik (*ang. driver*) i jest tworzony dla każdego formatu danych z osobna. Przede wszystkim dostarcza narzędzi do odczytu i zapisu danych geoprzestrzennych, pozwala na odczyt, zapis i tworzenie zbiorów danych i definiuje implementację warstwy w zakresie zależnym od sterownika.

Należy za każdym razem zapoznać się z implementacją sterownika dla danego formatu danych geoprzestrzennych oraz ograniczeń jaki dany format nakłada, zwłaszcza w zakresie obsługiwanych typów i nazw pól rodzaju obsługiwanych geometrii, nazw kolumn geometrii, typu identyfikatora elementu (*ang. feature ID*) czy innych specyficznych ograniczeń.

1.3.1 Zbiór danych wektorowych

Dane wektorowe są zorganizowane w postaci zbiorów. W zależności od formatu zapisu danych zbiór danych może zawierać tylko jedną lub dopuszczać więcej warstw. W wektorowym modelu danych wszystkie operacje wykonuje się na warstwach, a zbiór danych jest jedynie abstrakcyjną strukturą reprezentującą źródło danych: pojedynczy plik, zbiór plików, katalog lub połączenie z bazą danych. Zbiór danych zawiera listę warstw i udo-

stępniania odwołań do nich. W przypadku, gdy źródłem danych są standardowe bazy SQL zbiór danych udostępnia metodę umożliwiającą wykonanie zapytania SQL i pozyskania jedynie podzbioru danych jako źródła. Warstwa lub warstwy zawierają elementy (features), te z kolei zawierają geometrię elementu i przypisane do elementu atrybuty.

Warstwa i klasa elementów Warstwa w zbiorze danych wektorowych definiuje zbiór elementów należące do jednej klasy. Klasa elementów zdefiniowana jest niezależnie od warstwy, pomimo, że relacja pomiędzy warstwą ja jej klasą elementów jest 1:1. Przyczyna tego stanu rzeczy ma swoje podstawy w definicji *simple feature* i nie będzie tu szczegółowo omawiana. Niemniej jednak jest przyczyną zamieszania i uniemożliwia wykonanie opisu struktury modelu w sposób jednoznacznie przejrzysty. Warstwa jest modelem abstrakcyjnym a jej każdorazowa implementacja zależy od tego jaki sterownik jest używany do odczytu i zapisu danych. W wektorowym modelu danych klasa elementów określana jest jako *FeatureDefn* i zawiera przede wszystkim informacje na temat:

- **zasięgu warstwy**: zasięg może być definiowany na podstawie maksymalnej rozpiętości geometrii elementów warstwy lub może być zdefiniowany niezależnie;
- **typów i nazw atrybutów**: są one definiowane w sposób podobny do definiowania pól w bazach danych. Należy pamiętać o ograniczeniach nakładanych przez sterownik docelowego formatu zapisu danych (np. długości nazw), które można znaleźć w dokumentacji sterownika. Należy upewnić się co do obsługiwanych typów pól dla danego formatu.
- **liczności elementów w warstwie**
- **georeferencji**: zdefiniowanej dla całości warstwy jako łańcuch WKT2, podobnie jak dla modelu wektorowego

Element(feature) Jest to reprezentacja pojedynczego obiektu w zbiorze danych geoprzestrzennych i jest podstawowym elementem modelu danych wektorowych. Element składa się z unikalnego identyfikatora, przechowywanego zwykle jako int64, geometrii oraz wartości atrybutów. Dodatkowo przechowuje informację o klasie geometrii, jeżeli ta nie jest zdefiniowana w warstwie oraz georeferencji, będącej zwykle odwołaniem do georeferencji warstwy. Model danych wektorowych zezwala na powiązanie więcej niż jednej geometrii z daną feature, nie jest to jednak rozwiązanie akceptowane przez większość formatów geoprzestrzennych w związku z tym przyjmuje się że w danym elemencie istnieje tylko jedna geometria.

Geometria Wektorowy model danych z założenia obsługuje wszystkie typy geometrii zdefiniowane w standardzie OGC: Punkt (wkbPoint), Linię (wkbLineString), Poligon (wkbPolygon) oraz Kolekcję geometrii (wkbGeometryCollection). Ponadto obsługuje odmiany MulitPoint, MultiLineString i MultiPoligon (odpowiednio: wkbMultiPoint, wkbMultiLineSting, wkbMultiPolygon), czyli agregację kilku osobnych obiektów geometrycznych należących do jednej klasy ze wspólnym ID. Jedynie trzy pierwsze typy geometrii obsługiwane są przez wszystkie formaty geoprzestrzenne. Geometria jest przechowywana w formacie WKT lub WKB (well-known binary). Typ geometrii może być definiowany dla każdej feature osobno, lub może być definiowany globalnie dla całej warstwy. Jeżeli typ nie jest definiowany globalnie (jest ustawiony jako wkbUnknown) dowolny typ danych jest możliwy.

w ramach standardu OGC geometria reprezentowana jest jako przy pomocy standardu WKT - *well-known text* wspomnianego już przy okazji omawiania systemu odniesień geoprzestrzennych. WKT jest narzędziem wygodnym dla człowieka - pozwala zapisać i odczytać strukturę geometrii, niestety nie jest wygodny ani wystarczająco szybki dla obliczeń komputerowych - gdzie z reguły zastępowany jest przez swój binarny odpowiednik WKB - *well-known binary*.

WKT/WKB obsługuje szereg typów geometrii, z których tylko trzy (Point, LineString, Polygon) obsługiwane są przez wszystkie formaty danych geoprzestrzennych modelu wektorowego. Kolejne cztery: MulitPoint, MultiLineString i wkbMultiPolygon oraz GeometryCollection obsługiwane są prze wybrane formaty. Istotą tych formatów jest to że jedna lub więcej osobnych geometrii tworzy kolejkę posiadającą wspólny identyfikator. Dodatkowe formaty wymienione w standardzie OGC nie są obsługiwane lub są obsługiwane jednie przy pomocy niektórych narzędzi. Obiekty te określane są jako *Simple Features*.

Hierarchia geometrii przedstawia się następująco:

Geometria (Geometry) bazowa klasa dla wszystkich typów geometrii.

Punkt (Point) bezwymiarowa pojedyncza lokacja w przestrzeni. Każdy punkt posiada (X,Y) 3D (X,Y,Z) lub 4D (X,Y,Z,M), gdzie M odnosi się do dodatkowego atrybutu geometrii np. czasu. Punkt bez żadnych wymiarów to pusta geometria (EMPTY).

Krzywa (Curve) bezwymiarowa pojedyncza lokacja w przestrzeni. Każdy punkt posiada (X,Y) 3D (X,Y,Z) lub 4D (X,Y,Z,M), gdzie M odnosi się do dodatkowego atrybutu geometrii np. czasu. Punkt bez żadnych wymiarów to pusta geometria (EMPTY).

Polilinia (LineString) krzywa łącząca dwa lub więcej punktów w przestrzeni. Linia definiowana jest jako uporządkowany ciąg punktów XY, XYZ lub XY(Z)M. Pozycja punktów w ciągu definiuje przebieg linii.

Powierzchnia (Surface) podstawowy typ geometrii dla wszystkich typów dwuwymiarowych. Powierzchnie to obiekty posiadające obszar.

Polygon krzywych (CurvePolygon) powierzchnia płaska zdefiniowana przez 1 pierścień zewnętrzny i zero lub więcej pierścieni wewnętrznych. Standard OGC definiuje, że w pierścieniu zewnętrznym punkty są ułożone przeciwnie do ruchu wskazówek zegara (CCW) a w pierścieniach wewnętrznych w przeciwnym (CW). GDAL/OGR nie wymaga takiej kolejności.

Poligon (Polygon) ograniczona forma poligonu krzywych, gdzie każdy pierścień zdefiniowany jest jako polilinia (krzywa prosta.

Kolekcja geometrii (GeometryCollection) zbiór zera lub więcej instancji dowolnych geometrii.

Wielopowierzchnia (MultiSurface) ograniczona forma kolekcji geometrii, gdzie każda forma geometrii w kolekcji musi być typu powierzchnia.

Wielopoligon (MultiPolygon) ograniczona forma wielopowierzchni, gdzie każdy element w kolekcji musi być typu polygon.

Wielokrzywa (MultiCurve) ograniczona forma kolekcji geometrii, gdzie każda forma geometrii w kolekcji musi być typu krzywa.

Wielolinia (MultiLineString) ograniczona forma wielokrzywej, gdzie każdy element w kolekcji musi być typu polilinia.

Wielopunkt (MultiPoint) ograniczona forma kolekcji geometrii, gdzie każda forma geometrii w kolekcji musi być typu punkt.

1.3.2 Reprezentacja atrybutów w modelu wektorowym

Znaleść i sać

1.3.3 Sieciowy model danych

Sieciowy model danych (GNM - *geografic network model*) to odzwierciedlenie istniejęcych w rzeczywistości obiektów sieciowych (na przykład sieć drogowa). Model ten obecnie nie wydaje się w peł ni zaimplementowany w bibliotece GDAL. W praktyce jest to model wektorowy z dodatkową funkcjonalnością. Cechą sieci jest to że cechy przestrzenne i atrybutowe nie są rozdzielne. Funkcjonowanie sieci nie jest obsługiwane wewnątrz GDAL, ale przez zewnętrzne sterowniki. Geometria sieci jest przechowywana jako graf (lista par wierzchołków z dodatkowymi atrybutami), który przechowuje informację o wierzchołkach, kierunkach krawędzi, wagach krawędzi (koszt) itp.

1.4 Definiowanie pozycji geograficznej

Istnieją dwa sposoby określenia pozycji geograficznej dowolnego obiektu systemy odniesienia przestrznnego (*ang. coordinate reference systems* - CRS). Pierwszy sposób to podanie współrzędnych geograficznych - długości i szerokości geograficznej, który jest wspólny dla całej kuli ziemskiej i wymaga jedynie podania parametrów elipsoidy. Drugi sposób to układ w określonej projekcji definiowany w jednostkach długości (w metrach, rzadziej w stopach). Ta metoda wymaga dodatkowo podania rodzaju i parametrów rzutowania (projekcji) oraz punktu odniesienia, który jest początkiem układu współrzędnych. Pozycja w przestrzeni geograficznej zapisywana jest w postaci liczb, niemniej jednak model danych geoprzestrzennych nie wie jak interpretować poszczególne wartości. Z tego powodu CRS jest ważną częścią składową modeli danych geoprzestrzennych.

Well-known text - WKT Sposób zapisu CRS w modelach danych geoprzestrzennych jest definiowany przy pomocy standardu Well-known text v.2 (WKT2). WKT jest to format semistrukturalny, tagowany (podobnie jak XML), gdzie określonym tagom przypisywane są parametry. Poniższy przykładowy blok definicji WKT2 dla układu WGS 1984, czyli współcześnie stosowanego systemu dla współrzędnych geograficznych. Poszczególne tagi: np DATUM mają swoje parametry, które z kolei mogą być kolejnymi, zagnieżdżonymi tagami: na przykład DATUM ma parametr, który jest zdefiniowany jako SFEROID o określonej długości półosi i spłaszczeniu. Każdy z tagów może mieć parametr AUTHORITY, to jest odwołanie do jednej z baz danych systemów odniesienia przestrzennego, gdzie poszczególne układy odniesienia definiowane są przy pomocy kodu. Tu mamy odwołanie do systemu EPSG (bazy danych parametrów geodezyjnych), gdzie identyfikatorami poszczególnych systemów są liczby całkowite.

```
GEOGCS["WGS 84",
DATUM["WGS_1984",
SPHEROID["WGS 84",6378137,298.257223563,
AUTHORITY["EPSG","7030"]],
AUTHORITY["EPSG","6326"]],
PRIMEM["Greenwich",0,
AUTHORITY["EPSG","8901"]],
```

```
UNIT["degree",0.0174532925199433,
AUTHORITY["EPSG","9122"]],
AXIS["Latitude",NORTH],
AXIS["Longitude",EAST],
AUTHORITY["EPSG","4326"]]
```

Baza EPSG EPSG do baza danych parametrów geodezyjnych systemów projekcji. Skrót pochodzi od European Petroleum Survey Group, pierwszego twórcy tego zbioru. Każdej pozycji w bazie przypisany jest identyfikator - liczba całkowita pomiędzy 1024 a 32767, a parametry opisane są w systemie WKT. Baza cały czas jest cały czas zarządzana i aktualizowana przez IOGP Geomatics Committee. System EPSG jest wykorzystywany przez znakomitą większość projektów GIS - zarówno otwarto-źródłowych jak i własnościowych, zarówno do definiowania projekcji jak i transformacji danych z jednej projekcji do drugiej. Transformacja wykonywana jest przy pomocy systemu PROJ.

PROJ PROJ to biblioteka przeznaczona do wykonywania transformacji pomiędzy układami odniesienia przestrzennego. Pomiędzy 1994 a 2018 znana była jako PROJ.4 (wersja 4 biblioteki) od wersji 5 numer z nazwy biblioteki został usunięty. Od 2008 zarządzany przez OSGeo. Od wersji 6 PROJ do opisu i transformacji używa WKT, ale dawniej stosowany format PROJ4String jest także wspierany. PROJ4String nie jest zalecany, ponieważ zawiera mniej informacji niż WKT. Jest jednak bardziej zwarty i czytelniejszy dla człowieka.

1.5 Konwencja nazewnictwa metod, klas i funkcji biboteki GDAL

ć roz-

Rozdział 2

Dostęp do danych geoprzestrzennych

2.1 Reprezentacja danych geoprzestrzennych w Pythonie

numpy array referencje i kopie Driver czytanie

Przygotowanie środowiska pracy - osobny rozdział

2.2 Nawiązywanie połączenia ze źródłami danych geoprzestrzennych

Podstawowe pojęcia związane z dostępem do danych geoprzestrzennych to warstwa danych i źródło danych przestrzennych. Sama warstwa jest abstrakcyjnym pojęciem systemów informacji geograficznej i nie ma ścisłej definicji. Jest to po prostu sposób organizacji danych w geoprzestrzennej bazie danych i oznacza grupę obiektów należących do jednej klasy i reprezentujących tę samą klasę bytów świata rzeczywistego. Wszystkie składowe warstwy są umieszczone w jednolitym układzie odniesienia przestrzennego, mają jasno zdefiniowany zasięg oraz własności geometrii. Niestety w chwili opuszczenia środowiska SIG stajemy przez złożonym problemem pracy z źródłami danymi zorganizowanymi w bardzo zróżnicowany sposób. W zależności od formatu przechowywania, źródła danych mogą być udostępniane w postaci pojedynczych plików, zbiorów plików, złożonych katalogów lub też relacyjnych baz danych. Indywidualne obsłużenie wszystkich (prawie 300), a nawet części rozwiązań przechowywania danych jest zadaniem zbyt złożonym, aby realizować je w każdym projekcie z osobna. Z tego powodu podstawą programowania geoinformacyjnego jest abstrakcja źródeł danych a nie bezpośrednia praca każdym formatem danych przestrzennych z osobna. Istniejąca dyskowa struktura organizacji: pliki, katalogi czy bazy danych przekształcane są do postaci abstrakcyjnej poprzez odpowiedni sterownik. Taką strukturę dyskową nazywamy źródłem danych (*data source name* - DSN). W przypadku danych wektorowych, które z są dostosowane do pracy z systemami bazodanowymi, jedno źródło danych może przechowywać wiele warstw — każda z własnym zasięgiem, układem odniesienia i typem geometrii. Natomiast w modelach rastrowych, w znakomitej większości sterowników każda warstwa jest jednocześnie źródłem danych. Wynika to z faktu, że model rastrowy przechowuje dane w postaci plików obrazów, które nie wspierają modelu opartego na wielu niezależnych warstwach. Nie należy jednocześnie mylić z warstwami pasm zawartych w zbiorach wielopasmowych. Każde pasmo w pliku posiada tę samą roz-

dzielczość, zasięg i rozmiar, chociaż mogą różnić się typem danych. Porównując model rastrowy do wektorowego, pasmo jest raczej odpowiednikiem atrybutów. Tym samym nawiązanie połączenia ze źródłem danych wektorowych udostępnia nam jedynie listę warstw obecnych w źródle, natomiast połączenie ze źródłem danych rastrowych udostępnia nam wszystkie własności warstwy. Taki opis może być jednak mylący dla osób przyzwyczajonych do pracy z jednowarstwowymi źródłami danych – głównie popularnym formatem ESRI Shapefile, czy też bardziej ogólnie podejściem: jedno źródło – jedna warstwa. Tym zagadnieniem zajmiemy się szczegółowo w kolejnych częściach rozdziału.

Przed rozpoczęciem pracy, należy upewnić się czy zaimportowano odpowiednie biblioteki oraz czy ustawiono prawidłowo ścieżkę do katalogu z danymi.

,,

Listing 2.1: Ustawienie środowiska pracy

```
1 import os
2 import gdal
3 import numpy as np
4 import math
5
6 os.chdir(os.path.dirname(os.path.dirname(__file__)))
7 curDir = os.getcwd()
8 dataPath = os.path.join(curDir,"DATA")
```

Funkcje, metody i atrybuty. W programowaniu obiektowym rozróżniamy dwa sposoby wywoływania funkcji: jako funkcje w więc fragmenty kodu wywoływane przez nazwę, do których możemy przekazać dowolne dane zgodne z definicją funkcji oraz metody, które są funkcjami, ale połączonymi z obiektami, tym samym do funkcji w sposób niejawny przekazywany jest obiekt, z którego wywołano funkcję. W przypadku obiektów należy również odróżnić metody i atrybuty. Metody zmieniają stan wewnętrzny obiektów natomiast atrybuty to zmienne wywnętrz obiektów. Różnica składniowa między metodami a atrybutami jest taka, że atrybuty nie mają nawiasów.

Samo nawiązanie połączenia ze źródłem danych realizujemy przy pomocy funkcji `Open()`, która nawiązuje połączenie ze źródłem danych geoprzestrzennych. Nie należy mylić procedury nawiązywania połączenia, z procedurą wczytywania danych w programach użytkowych i niektórych językach programowania, która również określana jest terminem *open* lub jego wariantami. Zarówno samo udane nawiązanie połączenia, jak również porażka nie są sygnalizowane żadnym komunikatem. Przyczyną niepowodzenia może być zarówno brak źródła danych, nieprawidłowy typ danych jak i to że wskazany obiekt nie jest źródłem danych geoprzestrzennych. Jako źródła danych testowych użyjemy dwa źródła danych: Geopackage (GPKG) zawierający poligony oraz GeoTiff, zawierający model wysokościowy. W listingu **??** pokazano mechanizm nawiązywania połączenia ze wskazanymi źródłami danych. Aby uniknąć błędów należy w pierwszej kolejności sprawdzić, czy plik istnieje `os.path.isfile()`, a następnie zidentyfikować sterownik `gdal.IdentifyDriver()`

Listing 2.2: Źródło danych

```
1 vectFile = os.path.join(dataPath, "polygons_1992.gpkg")
2 rastFile = os.path.join(dataPath, "dem_poznan_30.tif")
3
```

```
4  if os.path.isfile(vectFile):
5          vectDriver = gdal.IdentifyDriver(vectFile)
6
7  if os.path.isfile(rastFile):
8          rastDriver = gdal.IdentifyDriver(rastFile)
9
10 print(rastDriver.ShortName)
11 print(vectDriver.ShortName)
```

Wynikiem działania kodu, powinna być informacja o sterownikach przy pomocy których GDAL realizuje dostęp do danych geoprzestrzennych:

```
>>> GTiff
>>> GPKG
```

metadan
rownika

Od wersji GDAL 2.0 nie jest zalecane używanie funkcji `gdal.Open()` i `ogr.Open()` w zamian, zalecane jest użycie `gdal.OpenEx()`, która pozwala otwierać zarówno źródła danych rastrowych i wektorowych. Listę wszystkich sterowników źródeł danych geoprzestrzennych zarejestrowanych w systemie można pozyskać przy pomocy prostego polecenia:

```
allDrivers = [gdal.GetDriver(i).ShortName for i in range(gdal.GetDriverCount())]}
```

Skuteczność nawiązania połączenia można zweryfikować porównując utworzony obiekt z **None**. Jeżeli nie jest to None należy zweryfikować ścieżkę dostępu do źródła danych przy pomocy metody `object.GetDescription()`.

Listing 2.3: Nawiązywanie połączenia

```
1  dsn = gdal.OpenEx(vectFile)
2  if dsn is not None:
3          dsn.GetDescription()
4
5  rastLayer = gdal.OpenEx(rastFile)
6  if rastLayer is not None:
7          rastLayer.GetDescription()
```

Pozostaje krok, który pozwoli odróżnić od siebie źródło danych wektorowych od rastrowych. Źródło danych wektorowych jedynie pozwala wyświetlić listę i liczbę warstw, natomiast źródło danych rastrowych — czyli tak na prawdę warstwa pozwala już określić geoprzestrzenne parametry warstwy. GDAL nie oferuje narzędzia, które pozwala odróżnić oba typy źródeł danych bezpośrednio, można to natomiast zrobić w sposób pośredni: metoda `object.GetLayerCount()` zwraca 0, w przypadku danych rastrowych lub liczbę warstw (1 i więcej) dla obiektu wektorowego. Natomiast atrybut `object.RasterCount` zwraca liczbę pasm, nie mniej niż 1 dla obiektów rastrowych lub 0 dla obiektów wektorowych. Jest jednak możliwość, że źródło danych przechowuje zarówno warstwy rastrowe jak i wektorowe (np. potrafią to niektóre rozszerzenia geoprzestrzenne do baz danych, w tym GPKG) w takiej sytuacji zarówno metoda `object.GetLayerCount()` jak i atrybut `object.RasterCount` zwrócą wartości większe od 0. Najlepiej cały mechanizm ująć w postaci funkcji. Wszystkie funkcje dostępne są również w kodzie dołączonym do podręcznika.

Listing 2.4: Sprawdzanie typu źródła danych

```
1  def getDataType(dataRef):
2          if dataRef is None:
3                  print("Cannot open data source")
```

```
4                   return 0
5           tp = 0
6           if dataRef.RasterCount > 0:
7                   tp +=1
8           if dataRef.GetLayerCount() > 0:
9                   tp +=2
10          return tp
11
12  getDataType(dsn)
13  getDataType(rastLayer)
```

```
>>> 2
>>> 1
```

Pokazana powyżej funkcja w sprytny sposób wykorzystuje kolejne flagi systemu binarnego do zakodowania wszystkich możliwych wariantów typu źródła danych. Funkcja zwróci 0 (równoważne `False`), jeżeli w zbiorze nie ma ani danych rastrowych ani wektorowych; 1 jeżeli są tyko dane rastrowe, 2 jeżeli tylko dane wektorowe oraz 3 jeżeli oba typy danych.

Pozycyjne kodowanie binarne. W przypadku kodowania informacji na zasadzie binarnego (tak/nie) występowania kilku zjawisk można użyć metody polegającej na przypisaniu zjawisk do określonej pozycji w sekwencji bitów a następnie odczytać tę sekwencję jako zwykłą liczbę. W naszym przypadku schemat kodowania wygląda następująco:

```
brak warstw danych |0|0|0| = 0
tylko raster       |0|0|1| = 1
tylko wektor       |0|1|0| = 2
oba typy danych    |0|1|1| = 3
```

Oznacza to, że każda unikalna wartość (w tym wypadku od 0 do 7) koduje unikalną mieszankę wystąpień opisywanych zjawisk. Użytkownicy systemu linux, powinni zauważyć podobieństwo do szybkiego nadawania uprawnień użytkownikom w systemie plików. Jest to ten sam mechanizm.

2.3 Model danych rastrowych – czytanie i właściwości danych

Model danych rastrowych nie jest zdefiniowany w formie jednolitego standardu, gdyż większość formatów danych nie jest w stanie obsługiwać wszystkich dostępnych opcji. Model danych rastrowych opisany w oryginalnym podręczniku GDAL: https://gdal.org/user/raster_data_model.html jest luźno oparty na OpenGIS Grid Coverages specification (nie jest to standard Open Geospatial Consorcium).

Model danych zakłada że zbiór danych (*dataset*) to połączenie jednej lub więcej warstw rastrowych oraz informacji odnoszących się do całości zbioru. Każda warstwa rastrowa (*raster band*) posiada dodatkowo swój własny zbiór informacji, odnoszący się wyłącznie do tej warstwy. Dodatkowo, zarówno cały zbiór danych jak i poszczególne warstwy mogą posiadać metadane w postaci par: nazwa - wartość. Opis zawarty na stronach podręcznika definiuje wszystkie możliwe informacje zawarte w zbiorze danych rastrowych, jednakże

większość z nich ma charakter wysoce specjalistyczny - odnosi się do bardzo wąsko definiowanych źródeł danych i nie będzie tu omawiana.

Jeżeli nawiązano połączenie ze źródłem danych rastrowych, i liczba warstw rastrowych jest większa niż 0 (z reguły 1), można przystąpić do analizy podstawowych parametrów warstwy. Są to rozmiary: X i Y oraz poznana już wcześniej liczba pasm (*bands*). Do prezentacji użyjemy ponownie zbioru danych **dem_poznan_30.tif**.

Listing 2.5: Własności danych rastrowych

```
rastFile = os.path.join(dataPath, "dem_poznan_30.tif")
rastLayer = gdal.OpenEx(rastFile)
X,Y,rCount = rastLayer.RasterXSize,rastLayer.RasterYSize,rastLayer.
    RasterCount
print(X,Y,rCount)
```

```
>>> 785 839 1
```

2.3.1 Zasięg warstwy w przestrzeni geograficznej

Zasięg warstwy rastrowej nie jest odczytywany bezpośrednio, ale za pomocą geotransformaty. Geotransformata to zestaw współczynników transformacji afinicznej pozwalają przeliczyć dowolne położenie w obszarze macierzy danych na współrzędne geograficzne.

$$X_{geo} = GT_0 + Xpixel * GT_1 + Yline * GT_2$$

$$Y_{geo} = GT_3 + Xpixel * GT_4 + Yline * GT_5$$

Gdzie: GT_0 i GT_3 odpowiednio współrzędne X i Y lewego górnego narożnika macierzy danych; GT_1 i GT_5 odpowiednio rozpiętość komórki siatki w poziomie i w pionie, GT_5 jest z reguły ujemna; dla danych zorientowanych w kierunku północnym GT_2 i GT GT_4 wynoszą 0. Wartości różne od 0 są stosowane jedynie w przypadku zbiorów danych o innej orientacji i oznaczają odpowiednio przesunięcie wiersza lub komórki w wierszu o pewną wartość.

Pozyskanie zasięgu danych rastrowych polega na przeliczeniu położenia lewego górnego (NW) oraz prawego dolnego (SE) na współrzędne geograficzne zgodne z układem odniesienia. Pomimo, że przeliczenie komórki na współrzędne oparte jest o relatywnie prostą formułę, GDAL dostarcza funkcję `ApplyGeoTransform()`. Geotransformaty muszą zostać wcześniej pobrane przy pomocy metody **object**`.GetGeoTransform()`.

Listing 2.6: Geotransformaty

```
gt = rastLayer.GetGeoTransform() # geotransformaty
print(gt)

gdal.ApplyGeoTransform(gt,22,44)
gdal.ApplyGeoTransform(gt,0,0)
gdal.ApplyGeoTransform(gt,X,Y)
```

```
>>> (345484.8332, 29.987261, 0.0, 518149.3682, 0.0, -29.982121)
```

Tym samym obliczenie zasięgu warstwy można zamknąć w postaci prostej funkcji. Proszę zwrócić uwagę, że na kolejność przekazywania danych do funkcji `ApplyGeoTransform()`: liczba kolumn liczna wierszy. Podobnie, proszę zwrócić uwagę na odwrócenie kolejności współpodrzędnych y, w pierwszej kolejności podajemy drugą współrzędną, tak aby zachować kolejność rosnącą: południe-północ, a nie zgodności ze sposobem organizacji danych w macierzy.

W macierzy ndarray czy w pracy z obrazami stosuje się najczęściej kolejność 1) wiersz, 2) kolumna. GDAL stosuje kolejność geograficzną: x, y – czyli najpierw kolumna potem wiersz.

Listing 2.7: Zasięg warstwy rastrowej

```
def getRasterExtent(raster):
        c,r = raster.RasterXSize,raster.RasterYSize
        gt = raster.GetGeoTransform()
        x0,y0 = gdal.ApplyGeoTransform(gt,0,0)
        x1,y1 = gdal.ApplyGeoTransform(gt,c,r)
        return x0, x1, y1, y0

extent = getRasterExtent(rastLayer)
print(extent)
```

```
>>> (345484.8332, 369024.8332, 492994.3682, 518149.3682)
```

Przeliczanie współrzędnych geograficznych na położenie komórki rastra jest również możliwe przy pomocy funkcji `ApplyGeoTransform()`, po wcześniejszym wyliczeniu transformaty odwrotnej. Należy zwrócić uwagę, że wiersz i kolumna zostały podane w postaci liczby rzeczywistej, gdzie część dziesiętna oznacza położenie punktu na obszarze komórki. Aby uzyskać numer wiersza i komórki, a więc wartości całkowite należy wartości zaokrąglić w dół, funkcją `int()` lub `math.floor()`).

Listing 2.8: Odwrotne geotransformaty

```
gtInv = gdal.InvGeoTransform(gt)

rc = gdal.ApplyGeoTransform(gtInv,346143, 516821)
r,c = rc[0],rc[1]
print(r,c)
```

```
>>> (21.948213169074734, 44.30534366130087)
```

2.3.2 Pobieranie wartości danych

Po nawiązaniu połączenia dane nie są automatycznie wczytywane do pamięci. Jest to oczywiste podejście, gdyż nie zawsze potrzebny jest cały zbiór danych, jak również, ze względu na objętość, nie zawsze załadowanie całego zbioru jest możliwe. Mechanizmy ograniczania wielkości zbioru ładowanych danych będą omówione w dalszej części podręcznika, w pierwszej kolejności zostaną przeanalizowane kwestie związane z ogólną charakterystyką danych: typem danych i statystykami. Dostęp do tej informacji jest możliwy jeszcze

przed rozpoczęciem ładowania danych i pozwala zdefiniować strategię ich przetwarzania. Aby pozyskać dane, w pierwszej kolejności należy określić pasmo, z którego będą ładowane dane. Złuży do tego funkcja **object**`.GetRasterBand(1)`, gdzie domyślnie jest to zawsze pierwsze pasmo. Dopiero po pozyskaniu referencji do pasma, możliwe jest wydobycie jego atrybutów.

Raster nie posiada pasma zerowego, numeracja pasm rozpoczyna się zawsze od 1.

Rodzaje danych z punktu widzenia systemów informacji geograficznej. Macierze danych rastrowych mogą przechowywać wyłącznie wartości numeryczne, ale złożoność typów danych, również prezentowana w niniejszej pracy jest istotna jedynie w kontekście zakresu wartości i wydajności obliczeniowej. Z punktu widzenia SIG dane rastrowe mogą pełnić dwie funkcje:

kategorie : mają charakter dyskretny i są to zawsze liczby całkowite nieujemne, gdzie każda liczba to osobna unikalna kategoria. Kolejność liczb może mieć znaczenie, ale nie musi. Kategorie zawsze zaczynają się od 1 a 0 to kategoria nieoznaczona. Do danych kategoryzacyjnych używa się typu bez znaku o głębi 8 (0-255) lub 16 (0-65536) możliwych kategorii.

Wartości : mają charakter ciągły i mogą być różnego typu, zarówno jako liczby całkowite jak i zmiennoprzecinkowe. Na przykład pojedyncze pasma kanałów zobrazowań lub obrazów RGB mogą być zapisane przy pomocy 8-bitowego typu bez znaku, ale liczby w komórkach pełnią rolę wartości a nie symboli kategorii. Głębia bitowa jedynie określa precyzję odwzorowania zjawiska. Oczywiście możliwości najwierniejszego odwzorowania rzeczywistości mają liczby zmiennoprzecinkowe podwójnej precyzji, ale odbywa się to kosztem wydajności obliczeniowej. Sposób zapisu wartości to zawsze kompromis między wydajnością obliczeń i objętością danych a precyzją odwzorowania wartości.

Określenie typu danych. Najważniejszym atrybutem jest typ danych. Na jego podstawie można dopiero określić funkcję warstwy. Typ danych rastrowych pobierany jest przy pomocy atrybutu `band.DataType`, który jednak zwraca liczbę całkowitą (stałą). W bibliotece GDAL do liczby przypisana jest bardziej zrozumiały symbol mnemotechniczny, rozpoczynający się od przedrostka **GDT_**. W tabeli 2.1 podane są wszystkie typy danych obsługiwane przez model rastrowy i ich odpowiedniki zarówno w języku C jak i w bibliotece numpy. Nie są wszystkie typy numeryczne obsługiwane przez GDAL są również obsługiwane przez tablice wielowymiarowe **numpy.ndarray** i na odwrót. Dotyczy to przede wszystkim zdefiniowanego w GDAL typu zespolonego całkowitego (CInt), który nie ma większego sensu matematycznego oraz typu int8, który nie jest obsługiwany przez GDAL. Typ ten dla odmiany nie ma większego sensu z punktu widzenia zakresu przyjmowanych wartości $[-127, 128]$ co oznacza że nie nadaje się zarówno do kodowania kategorii, jak i do obliczeń na liczbach całkowitych. Najmniejszym typem całkowitym obsługującym liczby ujemne jest int16. GDAL nie obsługuje również macierzy danych kodowanych wartościami całkowitymi o 64-o bitowej głębi wartości. Duże wartości, powyżej 4294967295, należy kodować przy pomocy liczb zmiennoprzecinkowych.

symbol gdal	wartość	typ C	typ np
GDT_Unknown	0		
GDT_Byte	1	uint8_t	np.uint8
nieobsługiwany		int8_t	np.int8
GDT_UInt16	2	uint16_t	np.uint16
GDT_Int16	3	int16_t	np.int16
GDT_UInt32	4	uint32_t	np.uint32
GDT_Int32	5	int32_t	np.int32
nieobsługiwany		uint64_t	np.uint64
nieobsługiwany		int64_t	np.int64
GDT_Float32	6	float	np.float32
GDT_Float64	7	double	np.float64
GDT_CInt16	8	niestandardowy	nieobsługiwany
GDT_CInt32	9	niestandardowy	nieobsługiwany
GDT_CFloat32	10	float complex	np.complex64
GDT_CFloat64	11	double complex	np.complex128
GDT_TypeCount	12		

Tabela 2.1: Typy danych obsługiwane przez model rastrowy GDAL i ich odpowiedniki numpy.ndarray.

GDT_Byte odpowiata typowi Unsigned Integer czyli unit8_t. W języku C typ Byte to Int8, który przez GDAL nie jest obsługiwany

Stosowanie stałych symboli, powoduje, że zdecydowanie łatwiej jest przypisać oczekiwaną wartość poprzez podaną stałą lub porównać z oczekiwaną stałą: `rastBand.DataType == gdal.GDT_Float32` niż pozyskać informację jaki typ danych kryje się pod daną wartością stałej. W tym celu można posłużyć się tabelą 2.1 lub listą obsługiwanych typów danych, która znajduje się również w dokumentacji API (*Application Programming Interface*) GDAL. Aby szybko uzyskać informację na temat stałych wykorzystywanych przez bibliotekę GDAL posłużymy się poniższą funkcją:

Listing 2.9: Funkcja pozyskiwania listy

```
1 def getGdalConstNames(key):
2         if not key.endswith('_'):
3         key+='_'
4         all_vars = gdal.gdalconst.__dict__
5         return {k: v for k, v in all_vars.items() if k.startswith(key
              )}
6
7 getGdalConstNames('GDT')
```

```
>>> {'GDT_Unknown': 0,...
```

Typ wyliczeniowy jest to mechanizm przypisywania zrozumiałych dla człowieka nazw symbolicznych, a więc w postaci łańcuchów znaków, do których przypisane są unikalne, stałe wartości liczbowe symbolizujące jakiś stan. W podstawowej wersji Pythona, typ wyliczeniowy nie jest obsługiwany (klasa **enum** jest jednak częścią biblioteki standardowej), jest natomiast standardową cechą języka C. Wartość zakodowana łańcuchem w typie wyliczeniowym może być obsługiwana również przypisaną typowi wartością, np: `x=gdal.GDT_Byte` jest równoważne `x=1`. Listę typów wyliczeniowych najczęściej kończy się symbolem liczności elementów w danym typie, standardowo również 0 przypisane jest wartości nieokreślonej. W języku Python wartości stałe przechowywane są najczęściej jako atrybuty klasy i dostęp do nich można zrealizować poprzez atrybut **object**`.__dict__`, zwracający listę wszystkich atrybutów zdefiniowanych w obiekcie danej klasy.

Statystyki zbioru danych . Statystki służą przede wszystkim do wstępnego zorientowania się co do charakteru danych i co istotne są wyliczane bez konieczności wcześniejszego załadowania danych do środowiska Pythona. Statystyki mogą być wyliczane jako przybliżone lub dokładne, mogą też być przechowywane w samym źródle danych. Do pobrania statystyk służy metoda `stats = rastBand.GetStatistics(True,True)`, której oba atrybuty są ustawione na `False` (w praktyce na 0 w związku z kompatybilnością z językiem C) co oznacza że nie są one ani na nowo wyliczane ani nie są dokładne. Pobranie statystyk z parametrami `True,True` wymusi ich i będą dokładne a nie przybliżone. Należy jednak uważać wymuszaniem przeliczania statystyk w przypadku dużych zbiorów danych, gdyż może to spowolnić obliczenia.

Drugim elementem pozwalającym zorientować się co do wewnętrzej struktury danych warstwy jest histogram. Jest to rozszerzenie statystyk i pozwala nie tylko poznać zakres danych ale też charakter ich rozkładu (jednorodny, normalny, skośny, logarytmiczny itp.) Do wyliczenia histogramu potrzebujemy informacji na temat zakresu wartości oraz określić liczbę przedziałów (*bins*) dla których zostanie zliczona liczba komórek. Alternatywnie dla statystyk można pozyskać informację o wartości maksymalnej i minimalnej dla pasma odpowiednio funkcjami `band.GetMaximum()` i `band.GetMinimum()`. Szerzej, kwestie związane z zastosowaniem histogramu zostaną omówione w rozdzie poświęconym wizualizacji **??**.

Listing 2.10: Statystyki i histogram

```
1 stats = rastBand.GetStatistics(True,True) # pierwszy arg accept
      approx, drugi przelicz
2 stats = dict(zip(("min","max","avg","sd"),stats))
3 maxVal = band.GetMaximum()
4 minVal = bandGetMinimum()
5 histogram = rastBand.GetHistogram(minVal,maxVal,10)
6 print(stats)
7 print(histogram)
```

```
>>> {'min': 47.914741516113, 'max': 151.43179321289, 'avg': 82.299689922913, 'sd':
>>> [1442, 2537, 6327, 12172, 5463, 1672, 454, 288, 52, 7] # histogram
```

Pobieranie macierzy wartości . Samo pobranie wartości realizuje się przy pomocy metody **object**`.ReadAsArray()`. Jest to funkcja właściwa dla języka Python i nie ma ona swojego odpowiednika w wersji C/C++ biblioteki GDAL. W wersji bezargumentowej

wczytuje caly zbiór danych do pamięci komputera. Należy jednak podkreślić, że pobranie całego zbioru danych czasami (w praktyce prawie zawsze) nie jest dobrym pomysłem, ponieważ takie zbiory mogą przekraczać dostępną pamięć. Funkcja `object.ReadAsArray()` pozwala ograniczyć ilość pobieranych danych wskazując początkową kolumnę i wiersz oraz liczbę kolumn i wierszy do pobrania. Polecenie: `rastValuesPart = rastBand.ReadAsArray (10,20,10,10)` wczyta do zmiennej `rastValuesPart` macierz o rozmiarach 10×10 komórek rozpoczynając od 10 kolumny i 20 wiersza.

Tak pobrane dane nie są jednak definiowane zasięgiem geograficznym i GDAL nie udostępnia metody, którą można zdefiniować zasięg okna przy pomocy współrzędnych geograficznych. W tym celu można jednak wykorzystać geotransformatę odwrotną, aby przeliczyć współrzędne definiujące zakres geograficzny na odpowiedni wiersz i kolumnę. Całość, ponownie jak w poprzednich przypadkach zostanie zamknięta w postaci użytecznej funkcji. Podane współrzędne zostaną zamienione na narożniki, lewy-górny i prawy-dolny a następnie te narożniki zostaną przeliczone na wiersz i kolumnę. Aby dopasować zasięg do topologii siatki rastra należy odpowiednio zaokrąglić wyniki geotransformacji: w dół dla lewego i dolnego (zachód i południe) zasięgu macierzy oraz w górę dla prawego i górnego (wschód i północ)

Listing 2.11: Definiowanie zasięgu geograficznego okna danych rastrowych

```
1  def extentToWindow(gtInv,W,E,S,N):
2          lu = gdal.ApplyGeoTransform(gtInv,W, N) # left upper
3          rb = gdal.ApplyGeoTransform(gtInv,E, S) # left upper
4          l = math.floor(lu[0])
5          u = math.floor(lu[1])
6          r = math.ceil(rb[0])
7          b = math.ceil(rb[1])
8          return l,u, r-l, b-u
9
10 shrinkedExtent = (349100,360150,502100,510100)
11 window = extentToWindow(gtInv,*shrinkedExtent)
12 rastValuesExtent = rastBand.ReadAsArray(*window)
```

Rozwijanie krotki. Z matematycznego punktu widzenia najważniejszą cechą funkcji jest to że może ona przyjąć dowolną liczbę argumentów ale zwraca tylko jedną wartość. Python i kilka innych języków programowania omijają to ograniczenie imitując możliwość zwracania wielu wartości poprzez mechanizm rozwijania krotek. W praktyce funkcja Pythona nie łamie zasady i zwraca zawsze pojedynczy obiekt, zwijając listę zwracanych wartości to pojednycznej krotki. Mechanizm rozwijania krotek polega na tym, że krotka wartości prawej strony jest przypisywana do zmiennych z lewej strony. Ten sam mechanizm jest stosowany również przy przekazywaniu argumentów do funkcji. Jeżeli funkcja posiada kilka argumentów, można do niej przekazać pojedynczą krotkę poprzedzoną znakiem *. W takiej sytuacji kolejne elementy krotki przypisywane są do kolejnych argumentów funkcji. W listingu 2.3.2 krotka ‘window‘ zawiera kolejno kolumnę, wiersz, liczbę kolumn, liczbę wierszy, które następnie przypisywane są odpowiednim argumentom funkcji `object.ReadAsArray()`. Bardziej zaawansowanym mechanizmem jest rozwijanie słowników, które dodatkowo pozwala na zdefiniowanie nazw argumentów.

Wartość pusta. W procesie zarządzania danymi geoprzestrzennymi występują lokalizacje, których wartość danej cechy nie jest ustalona. Brak informacji o wartości może mieć różne źródła: wartość cechy nie została ustalona, nie ma większego sensu (np. wysokość n.p.m na morzu, głębokość oceanu na lądzie itp.). Wartość pusta nie jest tożsama wartości 0, ale jest wartością nieokreśloną. Liczby nie mają wartości która symbolizuje wartość pustą. Najczęściej do symbolizowania wartości pustej wybiera się jedną liczbę, której pojawienie się w zbiorze danych jest niemożliwe i markuje się ją w nagłówku zbioru danych jako wartość pustą. Najczęściej są to: minimalna możliwa wartość dla danego typu liczby zmiennoprzecinkowej, wartość not-a-number lub jakaś wartość, której pojawienie się nie jest możliwe, np: -9999 dla wartości wysokości na lądzie. Dla wartości całkowitych, oznaczających kategorie, jest to najczęściej 0. Drugim sposobem jest tworzenie osobnej warstwy - maski, która zaznacza przy pomocy wartości logicznych `True/False` czy w danym miejscu wartość występuje czy nie. Poza wartościami pustymi, określanymi jako ‘na‘ - *not avialable*, występują jeszcze dwa rodzaje wartości nieokreślonych: ‘nan‘ - *not-a-number* np efekt wyciągania pierwiastka drugiego stopnia z liczby ujemnej w ciele liczb rzeczywistych, oraz ‘inf‘ - najczęściej w zaawansowanych systemach numerycznych efekt dzielenia przez 0 lub liczbę bardzo bliską zeru. Pozyskanie wartości pustej jest wymaga jej pobrania metodą `rastBand.GetNoDataValue()`. Należy jednak ustalić, czy wartość pusta (*nodata*), jest symbolizowana przez liczbę czy też przez jedną z wartości specjalnych.

Identyfikacja not-a-number. W Pythonie, podobnie jak w innych językach programowania ustalenie czy zmienna numeryczna przechowuje wartość czy nie wykonuje się testem: `nodata==nodata`. Zwrócenie wartości `True` oznacza że zmienna jest liczbą, natomiast `False` oznacza że jest liczbą nie jest.

W przypadku testowego zbioru **dem_poznan_30.tif** wartość pusta jest wyrażona nieliczbą: ‘np.nan‘

W celu ujednolicenia przyszłego procesu przetwarzania danych, aby uniknąć ewentualnej ręcznej obsługi różnych wartości pustych warto zamaskować macierz przy pomocy pod-modułu numpy: **masked array**. Pozwala on maskować wskazane wartości i traktować je jako wartości puste. W zależności od tego czy wartość pusta reprezentowana jest przez liczbę, czy nie liczbę stosujemy różne metody maskowania: dla liczby `np.ma.masked_equal()` dla nie-liczby `np.ma.masked_invalid()`. Całość jak zwykle zamkniemy w postaci użytecznej funkcji:

```
1 def maskArray(array,nodata,copy=True):
2         if nodata == nodata:
3                 maskedValues = np.ma.masked_equal(array,nodata,copy)
                      # values
4         else:
5                 maskedValues = np.ma.masked_invalid(array,copy) # nan
6         return maskedValues
7
8 maskedRast = maskArray(rastValues,nodata)
```

Proces maskowania danych domyślnie powoduje powstanie kopii (nowego zbioru) a nie referencji do danych. Jeżeli nie chcemy tworzyć kopii a jedynie zamaskować istniejący zbiór, należy użyć argumentu `copy=False`, co może być konieczne w przypadku dużych zbiorów danych. Należy w takiej sytuacji pamiętać, że modyfikując wartość w referencji, modyfikujemy też oryginalne dane.

2.4 Czytanie i właściwości danych wektorowych

Jeżeli skutecznie nawiązano połączenie ze źródłem danych wektorowych i liczba warstw jest większa niż 0 można dokonać przeglądu dostępnych warstw. Warstwy wektorowe mają często swoje nazwy i jeżeli źródło danych zawiera wiele warstw, często wygodniejszym sposobem jest pobranie warstwy przez jej nazwę a nie numer porządkowy.

Warstwy wektorowe numerowane są od 0, nie należy mylić ze schematem numerowania pasm w rastrowym zbiorze danych

Listing 2.12: Pobieranie warstw wektorowych

```
1  dsn = gdal.OpenEx(vectFile)
2  dsn.GetLayerCount()
3  layersNames = [dsn.GetLayer(i).GetName() for i in range(dsn.
       GetLayerCount())]
4  print(layersNames)
5
6  vectLayer = dsn.GetLayer() # default 0
7  vectLayer = dsn.GetLayerByIndex(0)
8  print(vectLayer.GetName())
9  vectLayer = dsn.GetLayerByName('obreby')
10 print(vectLayer.GetName())
```

```
>>> ['obreby', 'polygons_1992']
>>> obreby
>>> obreby
```

Właściwości warstwy . Dane wektorowe zorganizowane są w postaci bardzo złożonej i hierarchicznej struktury. Ta struktura to lista elementów (*features*), które w obrębie jednej warstwy powinny należeć do tego samego typu geometrii i posiadać taki sam zestaw atrybutów. Rolą geometrii jest umiejscowienie obiektu w przestrzeni geograficznej, rolą atrybutów prezentacja różnych właściwości obiektu. Typ geometrii oraz lista atrybutów są zdefiniowane na poziomie klasy, określonej jako definicja warstwy (*layer definition*). Dostęp do obiektu uzyskujemy przy pomocy metody `layer.GetLayerDefn()` Ze względu na dziedziczenie dostęp do niektórych cech definicji warstwy, takich jak typ geometrii można zrealizować już z poziomu samej warstwy bez konieczności pozyskiwania jej definicji. Dla samej warstwy są natomiast są charakterystyczne liczba elementów (*features*) oraz zasięg warstwy. Zarówno definicja warstwy jak i liczba elementów są niezbędne do odczytania danych. Porównując strukturę warstwy z modelem bazodanowym, definicję warstwy możemy przyrównać do definicji struktury tabeli (definicja struktury danych) natomiast elementy do rekordów tej tabeli (rzeczywiste dane). Nie jest to przypadek: relacyjne bazy danych są często wykorzystywane do przechowywania danych wektorowych (zob. **??**).

```
1  layerDefn = vectLayer.GetLayerDefn()
2  geomType = layerDefn.GetGeomType()
3  nFields = layerDefn.GetFieldCount()
4  nGeoms = layerDefn.GetGeomFieldCount()
5  print(geomType,nFields,nGeoms)
6  nFeatures = vectLayer.GetFeatureCount()
```

```
>>> 6 2 1
```

Podobnie jak w przypadku typu danych rastrowych ustalenie typu geometrii danych wektorowych wymaga rozpoznania jaki typ geometrii symbolizuje dana liczba całkowita (w przykładzie 2.4 wartość 6). GDAL/OGR obsługuje wiele formatów geometrii, które w praktyce można sprowadzić do siedmiu podstawowych (punkt, linia, poligon, wielopunkt, wielolinia, wielopoligon, kolekcja geometrii), zaprezentowanych w tabeli 2.2. Szczegółowo własności poszczególnych geometrii zostaną omówione w dalszej części rozdziału. W zależności od tego czy punkty tworzące geometrię mają dodatkowe atrybuty Z (rzędna) i M (odległość wzdłuż trasy) są one definiowane odpowiednimi stałymi. Listę wartości poszczególnych stałych można pozyskać funkcją podobną do tej przedstawionej w listingu: 2.3.2 lub kolejnej która podaje mnemotechniczny symbol przypisany danej liczbie całkowitej.

```
1  def getOgrConstNames(key):
2          all_vars = gdal.ogr.__dict__
3          return {k: v for k, v in all_vars.items() if k.startswith(key
               )}
4
5  getOgrConstNames('wkb')
6
7  def ogrGetGeomTypeName(geom_type): # numer
8          types = getOgrConstNames('wkb')
9          index = list(types.values()).index(geom_type)
10         return list(types.keys())[index]
```

```
>>> {'wkb25DBit': -2147483648,
'wkb25Bit': -2147483648,
'wkbUnknown': 0,...
```

Pozostałe własności warstwy to liczba atrybutów (tu 2) i liczba pól geometrii – z reguły obsługiwane jest jedno pole, ale niektóre formaty oparte o relacyjne bazy danych umożliwiają przechowywanie w jednym rekordzie kilku geometrii.

wstawić
lar, być
poziomo

Tabela 2.2: Typy geometrii obsługiwane przez OGR

Atrybuty obiektów przechowywane w definicji warstwy odpowiadają ogólnie typom pól znanym z systemów baz danych. Każdy atrybut składa się z nazwy, typu danych i dodatkowych własności definiujących np długość łańcucha tekstu czy precyzję liczby. Generalnie obsługiwane typy pól to wartości całkowite (Integer), zmiennoprzecinkowe (Real), łańcuchy znaków (String), Data i Czas oraz tzw BLOB-y czyli różne dane binarne. Niektóre formaty źródeł danych pozwalają w pojedynczym polu przechowywać nie tylko atrybuty atomowe (czyli niepodzielne) ale również listy wartości atomowych. Jednakże każdy typ ´zapisu danych wektorowych (czyli sterownik) może mieć ograniczoną listę typów. Należy sprawdzić w metadanych sterownika jakie typy pól są przez dany sterownik obsługiwane przy pomocy polecenia: `driver.GetMetadata()['DMD_CREATIONFIELDDATATYPES']`. W tabeli 2.3 pokazano jakie typy pól są obsługiwane przez wybrane popularne formaty przechowywania danych geoprzestrzennych.

Kod	Kod liczb.	ESRI Shp.	GPKG	PostGIS	GeoJSON
OFTIntegerList	1	x	x	x	x
OFTReal	2	x	x	x	x
OFTRealList	3			x	x
OFTString	4	x	x	x	x
OFTStringList	5			x	x
OFTWideString	6				
OFTWideStringList	7				
OFTBinary	8		x	x	
OFTDate	9	x	x	x	
OFTTime	10			x	
OFTDateTime	11	x	x	x	
OFTInteger64	12	x	x	x	x
OFTInteger64List	13			x	x

Tabela 2.3: Typy pól danych atrybutowych obsługiwane przez OGR i wybrane formaty źródeł danych

Dostęp do właściwości pola uzyskujemy poprzez klasę definicji pola **field defniniton**. Dla każdego pola dostęp do obiektu klasy uzyskujemy przy pomocy metody `layerDefn.GetFieldDefn(i)`, gdzie `i`, to kolejny indeks pola. Własności pola to nazwa, nazwa typu, długość (a właściwie szerokość, ang. *width*) stosowana dla łańcuchów znaków czy precyzja, stosowana dla liczb rzeczywistych.

```
fieldDefn = layerDefn.GetFieldDefn(0)
fieldDefn.GetName()
fieldDefn.GetTypeName()
fieldDefn.GetWidth() # alternatywnie
fieldDefn.GetPrecision()
fieldDefn.GetType() # sprawdzanie typu
fieldDefn.GetType() == gdal.ogr.OFTString
```

```
>>> String # name
>>> 254 # width
```

Ponownie, GDAL/OGR nie oferuje żadnego wygodnego narzędzia, które pozwoliło by pozyskać listę atrybutów wraz z ich nazwami. Lukę tę może wypełnić poniższa funkcja. Lista pól atrybutowych nie jest iteratorem, w związku z tym aby wydobyć wszystkie pola należy w pierwszej kolejności określić ich liczbę

```
def getFieldsDesc(layer):
        layerDefn = layer.GetLayerDefn()
        nFields = layerDefn.GetFieldCount()
        fNames = [layerDefn.GetFieldDefn(i).GetName() for i in range(
           nFields)]
        fTypes = [layerDefn.GetFieldDefn(i).GetTypeName() for i in
           range(nFields)]
        return dict(zip(fNames, fTypes))

getFieldsDesc(vectLayer)
```

```
>>> {'jpt_nazwa_': 'String', 'nazwa_id': 'Integer'}
```

Feature: własności geometrii i wartości atrybutów Feature ma własną geometrię i własne wartości atrybutów

Porządkowanie atrybutów, ID obiektu

2.5 Filtrowanie danych

2.5.1 Filtrowanie danych rastrowych

wielkość okna maskowanie danych

2.5.2 Filtrowanie danych wektorowych

filtr zasięgu filtr atrybutowy

Rozdział 3

Praca z systemami odniesienia geoprzestrzennego

3.1 Odczytywanie odniesienia geoprzestrzennego ze zbiorów danych

porównanie mechanizmów odczytu danych legacy-code dla GDAL różne reprezentacje

3.2 Obiekt odniesienia przestrzennego

tworzenie nowego obiektu

3.3 Reprojekcja geometrii

3.4 Określanie pozycji geograficznej dla danych rastrowych

geotransformaty sampling

Przykład: Próbkowanie warstwy rastrowej (1992) warstwą wektorową w WGS84

Rozdział 4

Wizualizacja danych geoprzestrzennych

4.1 Wizualizacja warstw rastrowych

Zacząć od omówienia imshow

4.2 Podstawowe elementy kompozycyjne mapy

dobór palety jako element kompozycyjny

4.3 Statystyczna analiza rastra i jego kategoryzacja

4.4 Wyświetlenie rastra z wbudowaną paletą barwną

4.5 Wyświetlanie obiektów wektorowych

4.6 Nakładanie obiektów, definiowanie przeźroczystości

4.7 Przybliżenia

4.8 Wizualizacje 3D

Rozdział 5

Tworzenie nowych zbiorów danych geoprzestrzennych

5.1 Tworzenie nowego zbioru danych geoprzestrzennych

rola drivera specjalny driver mem

5.2 Tworzenie rastra

Jakie informacje są potrzebne Tworzenie rastra na podstawie innego rastra zarówno copy jak i używanie parametrów Opisać że mechanizm createcopy ma bardzo małe zastosowanie, w zasadzie jedynie do kopiowania danych do innego zbioru

Przykład: Tworzenie rastra z wartościami indeksów komórek

Przykład: Reprojekcja fragmentu warstwy rastrowej

gdal.Warp pracuje na całym zbiorze danych jeżeli chcemy reprojekcję fragmentu trzeba go wyciąć ReprojectImage obsolate

5.3 Tworzenie nowej warstwy wektorowej

5.4 Tworzenie nowej warstwy wektorowej na postawie warstwy istniejącej

Zaznaczyć że nie jest zalecane modyfikowanie warstwy, chyba że absolutnie wiemy co robimy Open z parameterm 1

Przykład: Tworzenie wektora na podstawie zbioru losowych danych punktowych

Zbiór zawiera trzy obiekty specjalne: - centroid - środek geometryczny - medoid (zmiana atrybutu istniejącego obiektu)

5.5 Tworzenie warstwy liniowej

5.6 Tworzenie warstwy poligonowej

Rozdział 6

Praca z geometrią liniową

6.1 Czytanie zbiorów danych liniowych

6.2 Własności geometrii linii

6.3 Tworzenie linii na podstawie pliku csv

Przykład: Tworzenie i wizualizacja przekroju na podstawie przebiegu linii

Rozdział 7

Praca z poligonami

7.1 Definicja geometrii poligonu

7.2 Czytanie geometrii poligonu

7.3 Obliczanie obwodu i powierzchni poligonu

7.4 Operacje na pojedynczych poligonach

convex hull, envelope, buffer, bounding

7.5 Wykorzystanie poligonu do filtrowania geometrii

7.6 Przycinanie poligonów

7.7 Mapy Woronoja

7.8 Tworzenie geometrii poligonu od podstaw

Rozdział 8

modyfikacja i tworzenie nowych geometrii

8.1 Konwersja pomiędzy punktami a liniami

Przykład: Utworzenie linii z losowo wygenerowanych punktów

Przykład: System hubów

8.2 Konwersja pomiędzy punktami a poligonami

Przykład: Utworzenie poligonu z punktów

8.3 Konwersja miedzy liniami i poligonami

8.4 Modyfikacja istniejących zbiorów danych

Przykład: Dociąganie punktów obiektów linijnych

Przykład: Upraszczanie geometrii linijnej

8.5 Wybrane obiekty proceduralne

Rozdział 9

Relacje i predykaty przestrzenne

Dissolve jako unia, monte carlo' topologia wykrywanie przecięć linii tworzenie atrybutów na podstawie nakładania rastrów (np udziały)

Przykład: Generowanie punktów w pewnej odległości od linii

to być może w następnym rozdziale

Rozdział 10

Łączenie zbiorów danych wektorowych

10.1 Łączenie dwóch zbiorów danych wektorowych

wymóg zgodności atrybutów

10.2 Dołączanie dodatkowych atrybutów do istniejącego wektora

Przykład: Modyfikacja wektora na podstawie analizy sąsiedztwa

10.3 łączenie zbiorów danych wektorowych o różnych atrybutach

Arbitralne decyzje jak wypełniać braki: - NULL - 0 - liczba porządkowa

10.4 Tworzenie złączeń między tabelami

10.5 Transfer atrybutów między warstwami wektorowymi

wymaga relacji przestrzennej

Rozdział 11

Konwersja pomiędzy typami danych: rastryzacja i wektoryzacja

maskowanie rastra wektorem? tu czy wyżej wektoryzacja linii i punktów poligony do punktów jako centroidy

Rozdział 12

Statystyki zonalne

Zrobić zarówno raster jak i wektor, rozpocząć od rastra

Rozdział 13

Algebra map i przetwarzanie dużych zbiorów danych

13.1 Operacje na wielu warstwach rastrowych

warunki konieczne algebra map w Pythonie

Łączenie danych https://www.mundialis.de/en/filter-raster-files-bounding-box-using-gdal/

13.2 Stosowanie metod rozmytych

Unikanie tworzenia dodatkowych warstw rastrowych: funkcja transformacji

13.3 Praca z dużymi zbiorami danych

Przykład: Obliczanie wskaźnika NDVI dla zdjęć Sentinel

13.4 Stosowanie kryteriów rozmytych na podstawie statystyk rastra

Rozdział 14

Operacje w oknach

14.1 Przetwarzanie zbiorów rastrowych w oknie

Jedna z podstawowych operacji na danych rastrowych są obliczenia w oknie. Obejmują one: statystyki centralne (obliczane dla otoczenia komórki), filtry i macierze splotowe (konwolucje) i bardziej zaawansowane przekształcenia jak parametry terenu. Język Python ze swojej natury nie jest dostosowany do przetwarzania danych rastrowych. Ze względu na słabą typowalność języka, tradycyjne pętle działają bardzo wolno. Z drugiej strony operacje wektorowe przy użyciu środowiska Scipy w przypadku pracy z pojedynczymi komórkami traci wydajność związaną z wektorowym przetwarzaniem danych. Biblioteki **numpy** i **scipy** oferują narzędzia wspomagające pracę z wybranymi technikami GIS i teledetekcji, takimi jak filtry czy operacje splotowe, jednak ich stosowalność jest ograniczona przede wszystkim brakiem obsługi wartości pustych, powszechnie występujących w zbiorach danych rastrowych. Wynika to z faktu, że w podstawowe narzędzia środowiska Scipy nie są przeznaczone bezpośrednio danych geoprzestrzennych, a jedynie tzw. obrazów naturalnych, gdzie wartości puste z reguły nie występują.

Aby zrozumieć mechanizmy przetwarzania danych rastrowych omówimy prosty filtr wygładzający (średnia) działający w oknie o dowolnej wielkości, przedstawimy mechanizmy obliczeń różnymi metodami:

- "naiwny" kod Pythona
- wbudowany filtr (**ndimage.uniform_fiter**) i zastosowanie filtra uniwersalnego
- operacje na wycinkach danych (*ang. slices*) - pełne wykorzystanie operacji wektorowych.

W pierwszej kolejności omówimy algorytm operacji: Zasada wykonywania obliczeń w

Algorithm 1 Liczenie sumy dla całego rastra

for all *cell* **do**
  Select $cell \pm radius$
  Mask $\emptyset \in values$
  $sum \leftarrow \sum \emptyset \notin values$
  $cell \leftarrow sum$
end for

ruchomym oknie jest prosta. Potrzebna jest procedura - najczęściej funkcja, która dla przekazanej porcji (wycinka) danych wylicza jedną wartość wynikową. Następnie procedura ta jest stosowana wiersz po wierszu dla wszystkich komórek zbioru rastrowego, a wyniki są zapisywane w nowym zbiorze na tych samych pozycjach co centralna komórka rastra. Mechanizmy przemieszczania się po zbiorze danych są zależne od narzędzia, a nawet od przyjętej technologii w ramach tego samego języka programowania.

14.1.1 Rozwiązanie naiwne

Naiwny kod w Pythonie to taki, który realizuje algorytm przy użyciu podstawowych narzędzi języka, typowy raczej dla języków proceduralnych niższego poziomu jak c/c++, c# czy Java.

Pierwsze rozwiązanie naiwne, to podstawowa funkcja przeliczająca to po prostu zagnieżdżona pętla po wszystkich komórkach okna dla którego liczona jest statystyka. W każdym kroku pętli, następuje sprawdzenie, czy wartość w komórce określonej zmiennymi r i c nie jest wartością pustą. Jeżeli wartość nie jest pusta, jest dodawana do zmiennej *suma*, a licznik wartości niepustych jest zwiększany o 1. Na końcu suma wartości jest dzielona przez liczbę niepustych i taka wartość jest zwracana jako wynik. Jeżeli liczba wartości niepustych wynosi 0, to zwracana jest wartość pusta (*np.nan*).

,,

Listing 14.1: Naiwny sposób wyliczenia wartości średniej dla komórki

```
n = 7//2 # radius a nie rozmiar
def cells_mean_loops_nan(array,row,col,n):
    suma = 0
    count = 0
    for r in (row-n,row+n):
        for c in (col-n,col+n):
            if ~np.isnan(array[r,c]):
                suma += array[r,c]
                count +=1
    return np.nan if count ==0 else suma/count

means = np.full((array.shape[0]-(n*2),array.shape[1]-(n*2)),np.nan)
for row in range(n,array.shape[0]-n):
    for col in range(n,array.shape[1]-n):
        means[row-n,col-n] = cells_mean_loops_nan(array,row,col,n)
```

Drugie rozwiązanie wykorzystuje mechanizmy wektorowe numpy w celu pobrania wycinka zdefiniowanego współrzędnymi środkowego wiersza i kolumny z całej macierzy rastra. Następnie w wycinku maskowane są wartości puste i dla wektora danych niepustych bezpośrednio wylicza wartość średnią, która zostaje zwrócona przez funkcję.

,,

Listing 14.2: Naiwny sposób wyliczenia wartości średniej dla komórki - z użyciem mechanizmów numpy

```
n = 7//2 # radius a nie rozmiar
def cells_mean_nan(array,row,col,n):
    window = slice(row-n,row+n),slice(col-n,col+n)
    values = array[window].flatten()
    return np.mean(values[~np.isnan(values)])
```

```
6
7  means = np.full((array.shape[0]-(n*2),array.shape[1]-(n*2)),np.nan)
8  for row in range(n,array.shape[0]-n):
9      for col in range(n,array.shape[1]-n):
10         means[row-n,col-n] = cells_mean_nan(array,row,col,n)
```

Mechanizm przetwarzania rastra dla obu rozwiązań to dwie pętle: pętla po wierszach i zagnieżdżona w niej pętla po komórkach w wierszu. Każdy krok pętli pobiera współrzędne wiersza i kolumny i przekazuje do funkcji, która zwraca wartość średnią dla okna. W ostatnim kroku pętla zapisuje wynik w nowym zbiorze rastrowym, który został utworzony jeszcze przed uruchomieniem pętli.

Zaletą takiego podejścia, zwłaszcza rozwiązania pierwszego jest bardzo duża swoboda w implementacji algorytmu, gdyż mamy możliwość kontroli każdego kroku przetwarzania danych. Wadą jest jego wydajność. Dla zbioru o rozmiarach 785 × 839 komórek referencyjny czas obliczeń wynosi ok. 6-8 sekund. Należy dodać, że algorytm nie jest kompletny: wylicza wartości tylko dla komórek, które posiadają pełne otoczenie oraz zakłada dostępność całej macierzy (macierz nie jest doczytywana w w trakcie pracy pomiędzy wierszami). Takie podejście wybrano ze względu na kompatybilność z pozostałymi metodami.Co ciekawe, najbardziej "naiwne" rozwiązanie 1 jest nieznacznie szybsze niż nieco bardziej "pythoniczne" drugie, co prawdopodobnie wynika z faktu, że głównym obciążeniem algorytmu jest główna pętla, natomiast wewnątrz w funkcji, bardzo częste wywoływanie metod biblioteki numpy jest nieznacznie bardziej obciążające niż proste operacje arytmetyczne w pętli.

14.1.2 Wbudowane filtry modułu ndimage

Pakiet **scipy** posiada moduł **ndimage** pozawalający na przetwarzanie wielowymiarowych zbiorów macierzy, funkcją o podobnym działaniu jest procedura **ndimage.unifrom_filter**, jednakże jej zastosowanie pomimo bardzo dużej szybkości, nie przynosi spodziewanych rezultatów, ze względu na obecność wartości pustych, których ten filtr nie obsługuje. W wyniku zastosowania tego rozwiązania zwrócony macierz składa się wyłącznie z wartości pustych. Modyfikacją jest zastosowanie filtra uniwersalnego **ndimage.universal_filter** i wskazanie jako funkcji liczącej **np.mean** lub **np.nanmean**.

Funkcja uniwersalna *np.mean* oblicza średnią z zadanej macierzy wielowymiarowej, ale w sytuacji, gdy w macierzy znajduje się choćby jedna wartość pusta funkcja również zwróci wartość pustą. Z tego powodu ta funkcja nie nadaje się do danych, gdzie takie wartości występują. W praktyce jej zastosowanie spowoduje powstanie otoczek wokół obszarów zawierających wartość pustą i na granicach danych. Druga funkcja *np.nanmin*, nie ma takich ograniczeń, ale jej stosowanie okupione jest drastycznym spatkiem wydajności z 2-3s dla *np.mean* do 14-16s dla *np.argmin*, a więc nawet poniżej wydajności rozwiązań naiwnych. Dodatkowo tylko nieliczne funkcje statystyczne w numpy mają wbudowaną obsługę wartości pustych: *min, max, mean, stddev*, dlatego to podejście nie wydaje się warte polecenia.

funkcja uniwersalna (ufunc) to funkcja, która przeznaczona jest do pracy z wielowymiarowymi tablicami w sposób zwektorywowany, element po elemencie. Podstawowe wersje funkcji uniwersalnych przeznaczone są do pracy z wartościami skalarnymi, ale są zgeneralizowane tak, aby wartości skalarne mogły być zastępowane elementami wielowymiarowych tablic takimi jak wektory czy macierze. Zaletą funkcji uniwersalnych jest to że mogą być przekazywane jako argumenty do innych funkcji.

Podkreślić należy, że stosowanie funkcji uniwersalnych wraz z modułem **ndimage** jest jedynie adaptacją rozwiązania przeznaczonego do zupełnie innego typu danych, czyli zobrazowań, nie posiadających z założenia wartości pustych. W praktyce opracowanie takich funkcji dla narzędzi analiz geoprzestrzennych jest jednym z wyzwań społeczności Pythona.

14.1.3 Użycie wycinków

Wycinki (*ang. slices*) to trzeci sposób podejścia - paradoksalnie - najbardziej wydajny a przy tym najmniej oczywisty. Wychodząc ze sposobu wyliczania wartości wynikowej omówionej w przykładzie pierwszym, algorytm prowadzący do otrzymania średniej dla każdej komórki jest następujący:

Algorithm 2 Obliczanie zawartości pojedynczego okna

```
suma ← 0
suma ← 0
for all cells do
    if cell is not null then
        suma+ = cell
        licznik+ = 1
    end if
end for
if licznik == 0 then
    wynik ← null
else
    wynik ← suma/licznik
end if
     return wynik
```

Podstawowym problemem wydajnościowym Pythona jest to że w każdym kroku natywnej pętli języka następuje sprawdzanie typu zmiennej a często wywoływanie całego bagażu własności obiektów. Jeżeli pętla **for** przechodzi po wszystkich komórkach rastra, w praktyce wielokrotnie, działanie takiego algorytmu jest po prostu wolne.

Rozdział 15

Przepróbkowanie i łączenie rastrów o różnych zasięgach

Łączenie rastrów zwykłe przed patch

Rozdział 16

Analizy w oknach o różnych rozdzielczościach danych wejściowych i wynikowych

analiza wzorców

Rozdział 17

Analizy obszarowe

analiza kosztów ruchu zlewnie i modelowanie hydrologiczne

Rozdział 18

Klasyfikacje nadzorowane i nienadzorowane danych geoprzestrzennych

statystyki wektorów i ich wizualizacja

Referencje

Bibliografia

Todo list