# 第10章 地理空間データの処理

地理空間データの処理や可視化方法

# 地理空間データの概要

# 地理空間データとは

地理空間データは、空間上の特定の地点または区域の位置を示す情報（位置情報）およびそれに関 連付けられたさまざまな情報を持つデータです。

次のような地理空間データがある

- 土地利用図（自然、災害、経済活動など）
- 地質図
- 主題図（ハザードマップなど）
- 都市計画図
- 地形図
- 統計情報
- 空中写真、衛星画像

# GIS

GISは、Geographic Information Systemの略で、地理情報システムとも呼ばれます。地理的位置か ら空間データ（位置に関するデータ）を加工し、分析や判断をするための技術です。地理空間データ を分析するうえでは、データを集約したり、可視化したりします。

日本政府は阪神・淡路大震災をきっかけに、国土空間データ基盤の整備しました。また、2007年には、GISを高度に活用できる社会の実現のために、地理空間情報活用推進基本法が制定されました。

# 地理空間情報活用推進基本法

「地理空間情報活用推進基本法」（平成19年法律第63号）は、平成19年5月23日に成立、5月30日に公布、8月29日に施行されました。

地理空間情報の活用推進に関する施策等を行う上では、以下のような事項を基本理念として実施することが必要であるとされています。（基本法第3条）

- 情報整備、人材育成、連携体制整備などの施策について、総合的・体系的に実施すること
- GIS、衛星測位の両施策による地理空間情報の高度活用の環境を整備することを目指すこと
- 信頼性の高い衛星測位によるサービスを、安定して利用できる環境を整備すること
- 国土の利用や整備等の推進、国民の生命や財産等の保護、行政運営の効率化・高度化、国民の利便性の向上、経済社会の活力の向上等に寄与する施策を講じること
- 民間事業者の能力の活用、個人の権利利益や国の安全等の保護に配慮して施策を講じること

https://www.gsi.go.jp/chirikukan/about_kihonhou.html

# 座標参照系（CRS）

- CRSは、Coordinate Reference Systemの略で、座標参照系とも呼ばれている。CRSは、GISにおける 位置情報のルールで、地理座標系と投影座標系の2種類がある

**EPSGコード**

EPSGコードは、European Petroleum Survey Groupコードの略で、GISで使用される要素（座標 系、投影法など）をまとめ、この集合体を区別するために割り振られたコードである

GISでCRSを設定 する際にEPSGコードがよく使われている

- European Petroleum Survey Group は石油探査に関連する応用測地学、測量学、地図学の専門家からなる欧州石油産業に関連する科学組織
- EPSG コードを確認するには、EPSG Geodetic Parameter Dataset（https://epsg.org/）などから検索可能

# データ形式

地理空間データのデータ形式は、主に下記の２つに分類される

- ベクターデータ
- ラスターデータ

# ベクターデータ

GISにおけるベクターデータとは下記の要素がある

ベクターデータの要素

| 要素 | 説明 |
| --- | --- |
| ポイント（点） | 点で位置情報を表現する。駅や県庁所在地など、狭い範囲の位置を示す場合に利用される |
| ライン（線） | 線で位置情報を表現する。道路や河川など、連続した位置を示す場合に利用されます。等高線や区 域などを分割する場合にも利用される |
| ポリゴン（多角形） | 面（立体）で位置情報を表現する。区画や海岸線など、囲まれた領域を示す場合に利用される |

# ラスターデータ

GISにおけるラスターデータは、行と列で構成されるセル（ピクセル）が格子状（グリッド）に構成さ れたデータ形式で、各セルには数値情報が含まれる

主に気象や衛星画像などに利用される。

ラ スターデータは、画像ファイルに格納され、このファイルを地理空間データとして扱うには、画像座標か ら地理座標系へ変換する。

ラスターデータは描画速度が高速にできる利点があるが、行政区画な どの明確な位置情報を必要とする場合はベクターデータを利用する

# GISで利用する主なライブラリ

## GISで利用するPythonの主なライブラリを紹介する

| ライブラリ | 概要 |
| --- | --- |
| GDAL | ベクターデータおよびラスターデータを扱えるライブラリ。多数のファイル形式に対応しており、データ 形式の変換も行える　https://gdal.org/ |
| Fiona | ベクターデータの読み込み／書き込みに対応したライブラリ https://fiona.readthedocs.io/ |
| GeoPandas | 地理空間データをpandasで操作できるライブラリ。ベクターデータの読み込みや書き込みも行える |
| Rasterio | ラスターデータの読み込み／書き込みに対応したライブラリ |
| Shapely | 地理空間データを操作／解析するライブラリ |
| Shapely | 投影法および座標変換するライブラリ。CRSの処理で使われる https://pyproj4.github.io/pyproj/ |
| Geopy | ジオコーディング（地理座標の操作）を行うためのライブラリ |
| folium | 地図に可視化を行うライブラリ。ベクターデータを再現して地図上に描画できる |
| leafmap | インタラクティブなマッピングと地理空間分析を行うためのPythonパッケージ https://leafmap.org/ |

# 地理空間データのファイル形式と読み込み

10-2-read-data.ipynb

# 地理空間データのファイル形式と読み込み

地理空間データでよく使われるファイル形式を、Pythonのオブジェクトとして読み込 む方法について解説する

- ベクターデータ
  - GeoJSON
  - シェープファイル
- ラスターデータ
  - GeoTIFF

# GeoJSON

GeoJSON形式は、JSONを利用したさまざまな地理空間データの構造を符号化するベクターデータのファイル形式でRFC7946で定められている

地理空間情報（点・線・面などの形状）はgeometry型として下記のデータ構造を持つ

- Point（点）
- LineString（線）
- Polygon（多角形）
- MultiPoint
- MultiLineString

The GeoJSON Format : https://datatracker.ietf.org/doc/html/rfc7946/

# GeoJSON

GeoJSONのデータ構造

```
{
  “type”: “FeatureCollection”,    ②
  "features": [
       {
         “type”: “Feature”,    ①
         “properties”: {    ④
              "name": "yurakucho"
         },
         “geometry”: {    ③
              "type": "Point",
              “coordinates”: [ 35.675056,   139.763333 ]
         }
       },
        : (省略)
    ]
}
```

Feature（①）は、空間的な要素を持つエンティティ

FeatureCollection（②）は、複数のFeatureのリスト

Featureには、geometry（③）およびproperties（④）のメンバーがある

geometryは空間の領域を格納し、Point（点）およびLineString（線）等のgeometry型を格納している

propertiesは、Featureが持つ属性を格納し、propertiesにはオブジェクトの名前や値などがあり、地図に描画する際のスタイリング情報が含まれる場合もある

地理座標系としてWorld Geodetic System1984を使用し、単位は10進度を使用

# GeoJSON

Shapelyのfrom_geojson()関数を実行してGeoJSONファイルを読み込む

```
from shapely import from_geojson

geojson = """{
    "type": "FeatureCollection",
    “features”: [ {
          "type": "Feature",
          "properties": {"name": "yurakucho"},
          "geometry": {"type": "Point","coordinates": [35.675056,139.763333]}
        },{
           "type": "Feature",
           "properties": {"name": "tokyo-kanda"},
            "geometry": {"type": "LineString","coordinates": [[35.681111,139.766667],[35.691667,139.770833]]}
        }
    ]
}"""

geo_collection = from_geojson(geojson)
```

```
type(geo_collection)
```

shapely.geometry.collection.GeometryCollection

# GeoJSON

to_geojson()関数を実行してJSONに出力。to_geojson()関数の引数indentを指定して、
出力を読みやすくしている

```
from shapely import to_geojson

print(to_geojson(geo_collection, indent=4))
```


```
{
    "type": "GeometryCollection",
    "geometries": [
        {
            "type": "Point",
            "coordinates": [
                35.675056,
                139.763333
            ]
        },
         : (省略)
    ]
}
```

# GeoJSON

GeoPandasのread_file()関数を実行してGeoJSONファイルをGeoDataFrameに読み込む
StringIOは変数geojsonのデータをファイル読込形式で読める用に変換している

```
from io import StringIO
import geopandas as gpd

geojson_gdf = gpd.read_file(StringIO(geojson))
geojson_gdf
```

| | name | geometry |
|---|---|---|
| 0 | yurakucho | POINT (35.67506 139.76333) |
| 1 | tokyo-kanda | LINESTRING (35.68111 139.76667, 35.69167 139.7... |

# GeoJSON

1行目の「geometry」列の要素の型を確認

```
type(geojson_gdf.loc[0, "geometry"])
```

shapely.geometry.point.Point

2行目の「geometry」列の要素の型を確認

```
type(geojson_gdf.loc[1, "geometry"])
```

shapely.geometry.linestring.LineString

1行目の「geometry」列を確認

```
geojson_gdf.loc[0, "geometry"]
```

2行目の「geometry」列を確認

```
geojson_gdf.loc[1, "geometry"]
```



# GeoJSON

GeoDataFrameのto_json()メソッドを実行してJSONに出力

```
print(geojson_gdf.to_json(indent=4))
```


```
{
    "type": "FeatureCollection",
    "features": [
        {
            "id": "0",
            "type": "Feature",
            "properties": {
                "name": "yurakucho"
            },
            "geometry": {
                "type": "Point",
                "coordinates": [
                    35.675056,
                    139.763333
                ]
            }
        },
        : (省略)
    ]
}
```

# GeoJSON

GeoDataFrameのto_file()メソッドを実行してGeoJSONファイルに書き出す。
GeoJSON形式に出力する場合は、to_file()メソッドの引数driverに"GeoJSON"を渡す。

```
geojson_gdf.to_file(
    "./data/yurakucho-kanda.geojson",
    driver="GeoJSON",
)
```

# シェープファイル

シェープファイル(Shapefile)は、ESRI社（Environmental Systems Research Institute, Inc.）が提唱したベク ター形式のファイルである。業界の標準的なフォーマットであるが、近年は他のフォーマットのファイルを使うことが好まれる

同じ名前(拡張子が異なる）の複数のファイルから構成されており、それぞれのファイルが異なる役割をもっている。

必須ファイルは下記の3ファイルで、オプションで.prj などの拡張子を持つファイルもある

GISを利用するアプリケーションで利用され、Pythonのパッ ケージでも扱える

シェープファイルの必須ファイルの拡張子

| 拡張子 | 説明 |
|---|---|
| .shp | 図形の形状データ（点、線、面）を格納するメインのファイル |
| .shx | 図形のインデックス情報（位置）を格納するファイル |
| .dbf | 図形に紐づく属性情報（名称、面積、人口など）を格納するファイル |

その他（推奨ファイル）
.prj: 図形が持つ座標系の定義情報を格納するファイル、正確な位置表示に不可欠である
.sbn, .sbx: 空間インデックスを格納し、データの高速な検索に役立つ
.cfg:使用した文字コードの識別コードページ
他

シェープファイルの拡張子: https://desktop.arcgis.com/ja/arcmap/latest/manage-data/shapefiles/shapefile-file-extensions.htm

# シェープファイル

シェープファイルを読み込む

```
shapefile_gdf = gpd.read_file("./data/yurakucho")
shapefile_gdf
```

| | name | geometry |
|---|---|---|
| 0 | yurakucho | POINT (35.67506 139.76333) |

シェープファイルの構成

yurakucho
- yurakucho.shp
- yurakucho.shx
- yurakucho.dbf
- yurakucho.prj
- yurakucho.cpg

to_file()メソッドを実行してシェープファイル（.cpg、.dbf、.prj、.shp、.shx） に書き出される。
to_file()メソッドの引数driverがNone（デフォルト）の場合は、シェープファ イルとして出力される

```
shapefile_gdf.to_file("./data/yurakucho2")
```

# GeoTIFF

- GeoTIFFは、地理画像データを共有するために利用されるラスターデータのファイル形式
- 地理タグ（地理空間のメタデータ）が付いたTIFF（Tagged Image File Format）画像ファイルの形式
- 仕様はOGC GeoTIFF Standardによって定義されている

OGC GeoTIFF Standard : https://docs.ogc.org/is/19-008r4/19-008r4.html

# GeoTIFF

rasterioをインストール

```
!pip install rasterio
```

Rasterioを利用して、GeoTIFFファイルをDatasetReaderオブジェクトに読み込む

```
import rasterio

dataset = rasterio.open("./data/L03-b-14_5236.tif")
```

ラスターデータは、画像の要素（ピクセル）が地理空間データにマッピングされている。bounds属性にアクセスすると、画像データにマッピングされている4隅の位置座標を取得できる

```
dataset.bounds
```

BoundingBox(left=136.0, bottom=34.6666666336, right=137.0, top=35.3333333)

データセットの座標参照系（CRS）を確認するには、crs属性にアクセスする

```
dataset.crs
```

RS.from_wkt('GEOGCS["JGD2000",DATUM["Japanese_Geodetic_Datum_2000",SPHEROID["GRS 1980",6378137,298.257221999999,AUTHORITY["EPSG","7019"]],AUTHORITY["EPSG","6612"]],PRIMEM["Greenwich",0],UNIT["degree",0.0174532925199433,AUTHORITY["EPSG","9122"]],AXIS["Latitude",NORTH],AXIS["Longitude",EAST],AUTHORITY["EPSG","4612"]]')

# GeoTIFF

ラスターデータは多次元のデータで、バンド（レイヤー）と呼ばれる2次元のデータが含まれてる。
複数のバンドを持つラスターデータもあり、バンドには1から順番にインデックス番号が振られている
インデックスを確認するには、indexes属性にアクセスする

```
dataset.indexes
```

(1,)

read()メソッドの引数にバンドのインデックスを渡して実行すると、該当するバンドのラスターデータを
numpy.ndarray型で取得できる

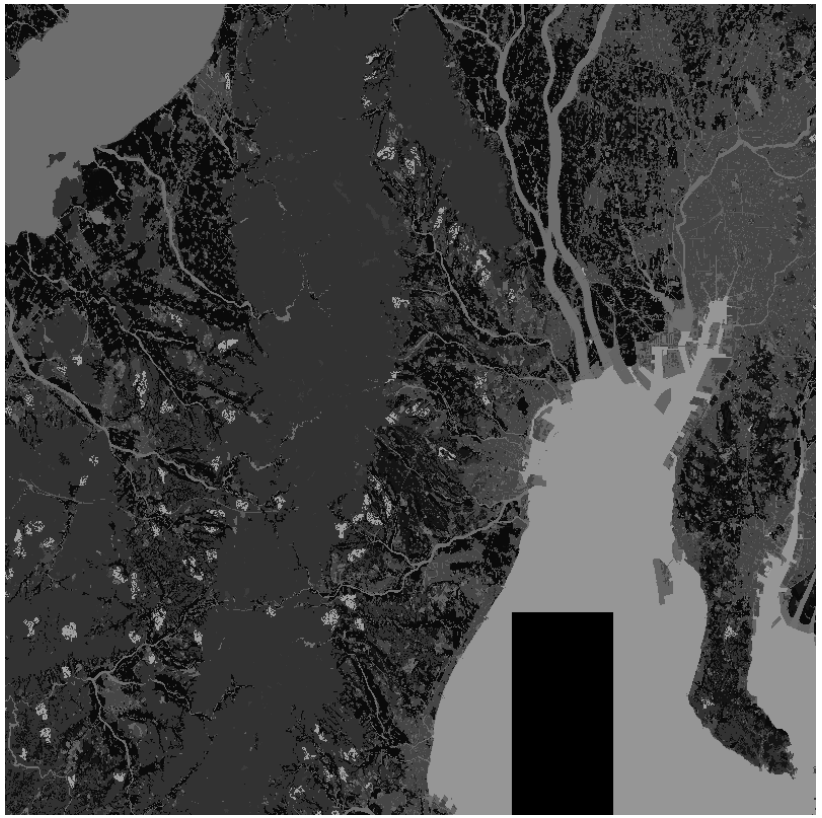
```
dataset.read(1)
```

```
ndarray (800, 800) hide data
array([[ 10,  10,  10, ...,  50,  50,  50],
       [ 10,  10,  10, ...,  50,  50,  50],
       [ 10,  10,  10, ...,  50,  50,  50],
       ...,
       [ 50,  50,  50, ..., 150, 150, 150],
       [ 50,  50,  50, ..., 150, 150, 150],
       [ 50,  50,  50, ..., 150, 150, 150]], dtype=uint8)
```

# GeoTIFF

pillowを使って画像データを描画する

```
from PIL import Image

Image.fromarray(dataset.read(1))
```



# 地理空間データの操作

10-3-operations.ipynb

# 地理空間データの操作

Shapelyで生成するGeometricオブジェクトを利用して、地理空間データを操作する方法を説明する

# Geometricオブジェクト

Geometricオブジェクトは、次のような地理空間データを
Pythonのオブジェクトとして処理できるインスタンスである

- Point（ポイント、点）
- LineString（ライン、線）
- Polygon（ポリゴン、多角形）
- MultiPoint
- MultiLineString

# Geometricオブジェクト

- ShapelyのPoint、LinearRing、Polygonクラスを利用してGeometricオブジェクトを生成 し、地理空間データを操作する
- ポイント（Point）を処理する場合は、PointクラスからGeometricオブジェクトを生成する
- 第1引 数にX値となる座標、第2引数にY値となる座標を渡すか、これらが含まれたシーケンス型のデータ を渡す

```
from shapely.geometry import Point

Point(1, 1)
# or
Point([1, 1])
```



地理的な座標を渡す場合は、経度と緯度の値を10進数で渡す

```
tokyo = Point(139.766667, 35.681111)
shinjuku = Point(139.700278, 35.690833)
shinagawa = Point(139.738694, 35.628222)
```

※ シーケンス型 https://docs.python.org/ja/3/library/stdtypes.html#sequence-types-list-tuple-range

# Geometricオブジェクト

LineStringを処理する場合は、LineStringクラスからGeometricオブジェクトを生成する。
引数には、ポイント（Point）を入れたシーケンス型を渡す。

```
from shapely.geometry import LineString

tokyo_shinjuku = LineString(
    [
        (139.766667, 35.681111),
        (139.700278, 35.690833),
    ]
)
tokyo_shinjuku
```

# Geometricオブジェクト

Polygonを処理する場合は、PolygonクラスからGeometricオブジェクトを生成する。
引数には、ポ イント（Point）を入れたシーケンス型のリストやタプルを渡す。

```
from shapely.geometry import Polygon

tokyo_shinjuku_shinagawa = Polygon(
    [
        (139.766667, 35.681111),
        (139.700278, 35.690833),
        (139.738694, 35.628222),
    ]
)
tokyo_shinjuku_shinagawa
```

# Geometricオブジェクトの属性

Geometricオブジェクトの型を確認する場合は、geom_type属性にアクセスする

```
tokyo.geom_type
```

Point

```
tokyo_shinjuku.geom_type
```

LineString

```
tokyo_shinjuku_shinagawa.geom_type
```

Polygon

Geometricオブジェクトの座標は、xyまたはcoords属性にアクセス

```
tokyo_shinjuku.xy
```

(array('d', [139.766667, 139.700278]), array('d', [35.681111, 35.690833]))

# Geometricオブジェクトの属性

coordsは、イテラブルオブジェクトです。リストに変換したり、スライスして値を確認できる

```
list(tokyo_shinjuku.coords)
```

[(139.766667, 35.681111), (139.700278, 35.690833)]

```
tokyo_shinjuku.coords[:]
```

[(139.766667, 35.681111), (139.700278, 35.690833)]

Geometricオブジェクトの長さを確認するには、length属性にアクセスする

```
tokyo_shinjuku.length
```

0.06709706852763346

```
tokyo_shinjuku_shinagawa.length
```

0.06709706852763346

# Geometricオブジェクトの属性

Geometricオブジェクトの面積を確認するには、area属性にアクセスする

```
tokyo_shinjuku_shinagawa.area
```

0.0018916006635003781

GeometricオブジェクトのX軸およびY軸の最小値、最大値を確認するには、bounds属性にアクセスする

```
tokyo_shinjuku.bounds
```

(139.700278, 35.681111, 139.766667, 35.690833)

```
tokyo_shinjuku.bounds
```

(139.700278, 35.628222, 139.766667, 35.690833)

# 距離の算出

Geometricオブジェクト間の距離を算出するには、distance()メソッドの引数にGeometricオブジェクトを渡して実行する

```
tokyo.distance(shinjuku)
```

0.06709706852763346

# 距離の算出

Geometricオブジェクトは、平面上の座標として処理する
地球が平面ではないことから、地理的な座標系（緯度経度）から距離などを算出するには、地球をモデル化したうえで処理する
GeoPyでは、地球を近似した楕円体モデル（WGS84）を利用してメートル法などで距離を算出できる

次のコードでは、geopyのdistance.geodesicクラスに2つの座標を渡して、2点間の距離を算出している。このクラスで座標は緯度, 経度の形式で渡すため、経度, 緯度であるPointインスタンスの順番をreversed()関数で反転している

```
from geopy import distance

tokyo_shinjuku_distance = distance.geodesic(
    list(reversed(tokyo.coords[0])),
    list(reversed(shinjuku.coords[0])),
)
tokyo_shinjuku_distance
```

Distance(6.105560821865678)

# 距離の算出

geodesicオブジェクトから、メートル（meters属性）やキロメートル（km属性）などの単位で距離 を取得できる

```
tokyo_shinjuku_distance.meters
```

6105.560821865678

```
tokyo_shinjuku_distance.km
```

6.105560821865678

# GeoPandas

10-4-geopandas.ipynb

# GeoPandas

- GeoPandasは、Pythonで地理空間データを簡単に操作できるPythonパッケージである
- pandasのデータ型を拡張して、形や位置などの空間データを扱えるようにした

# GeoDataFrame

- GeoDataFrameは、DataFrameを拡張したデータ型である
- 空間情報を持つGeoSeries型の列が含まれている
- GeoSeriesの要素はShapelyのGeometricオブジェクトである


図 10-1　GeoDataFrameの構造

# GeoDataFrame

read_file()関数を実行して、GeoJSONファイルをGeoDataFrameに読み込む

```
import geopandas as gpd

gdf = gpd.read_file("./data/N03-20220101_25_GML/N03-22_25_220101.geojson")
gdf.head()
```

| | N03_001 | N03_002 | N03_003 | N03_004 | N03_007 | geometry |
|---|---|---|---|---|---|---|
| 0 | 滋賀県 | None | None | None | None | POLYGON ((136.17602 35.70295, 136.17611 35.702... |
| 1 | 滋賀県 | None | None | 大津市 | 25201 | POLYGON ((135.88452 35.28393, 135.88484 35.283... |
| 2 | 滋賀県 | None | None | 彦根市 | 25202 | POLYGON ((136.24167 35.29628, 136.24408 35.295... |
| 3 | 滋賀県 | None | None | 長浜市 | 25203 | POLYGON ((136.17602 35.70295, 136.17611 35.702... |
| 4 | 滋賀県 | None | None | 近江八幡市 | 25204 | POLYGON ((136.09807 35.19687, 136.09816 35.196... |

# GeoDataFrame

シェープファイルを読み込む。N03-20220101_25_GMLディレクトリには、次のファイルが含まれており、これはGeoJSONファイルと同じデータとなる

- N03-22_25_220101.dbf
- N03-22_25_220101.prj
- N03-22_25_220101.shp
- N03-22_25_220101.shx
- N03-22_25_220101.xml

```
gdf = gpd.read_file("./data/N03-20220101_25_GML/")
gdf.head()
```

| | N03_001 | N03_002 | N03_003 | N03_004 | N03_007 | geometry |
|---|---|---|---|---|---|---|
| 0 | 滋賀県 | None | None | None | None | POLYGON ((136.17602 35.70295, 136.17611 35.702... |
| 1 | 滋賀県 | None | None | 大津市 | 25201 | POLYGON ((135.88452 35.28393, 135.88484 35.283... |
| 2 | 滋賀県 | None | None | 彦根市 | 25202 | POLYGON ((136.24167 35.29628, 136.24408 35.295... |
| 3 | 滋賀県 | None | None | 長浜市 | 25203 | POLYGON ((136.17602 35.70295, 136.17611 35.702... |
| 4 | 滋賀県 | None | None | 近江八幡市 | 25204 | POLYGON ((136.09807 35.19687, 136.09816 35.196... |

# .cxインデクサ

GeoDataFrameには、.cxインデクサという、空間座標に特化したインデクサがある。
添字を[経度, 緯度]の形式で指定でき、スライス記法も使える

次のコードでは経度が「136.2」から「136.4」、緯度が「35.5」から「35.6」の範囲のデータを抽出している

インデクサ：添字を用いてアクセスするためのもの

```
gdf.cx[136.2:136.4, 35.5:35.6]
```

| | N03_001 | N03_002 | N03_003 | N03_004 | N03_007 | geometry |
|---|---|---|---|---|---|---|
| **0** | 滋賀県 | None | None | None | None | POLYGON ((136.17602 35.70295, 136.17611 35.702... |
| **3** | 滋賀県 | None | None | 長浜市 | 25203 | POLYGON ((136.17602 35.70295, 136.17611 35.702... |
| **13** | 滋賀県 | None | None | 米原市 | 25214 | POLYGON ((136.36961 35.55895, 136.37013 35.558... |

# GeoSeriesの処理

地理空間データであるGeoSeriesは、「geometry」という列名（GeoJSONではgeometryメンバー）でgeometry型としてGeoDataFrameに追加される。それ以外の列（GeoJSONではpropertiesメン バー）は、pandasと同じデータ型で読み込まれる。

```
gdf.dtypes
```

| | 0 |
|---|---|
| **N03_001** | object |
| **N03_002** | object |
| **N03_003** | object |
| **N03_004** | object |
| **N03_007** | object |
| **geometry** | geometry |

**dtype:** object

# GeoSeriesの処理

GeoSeriesは、geometry型の要素を持ったSeriesである

```
geoseries = gdf.loc[:, "geometry"]
type(geoseries)
```

```
geopandas.geoseries.GeoSeries
```

GeoSeriesの要素は、Geometricオブジェクトになる。次のコードから、0番目の要素はPolygonであることが確認できる

```
type(geoseries[0])
```

```
shapely.geometry.polygon.Polygon
```

```
geoseries[0]
```

# GeoSeriesの処理

```
geoseries[0].geom_type
```

Polygon

GeoSeriesからもGeometricオブジェクトの要素へアクセスできる

```
geoseries.geom_type.head()
```

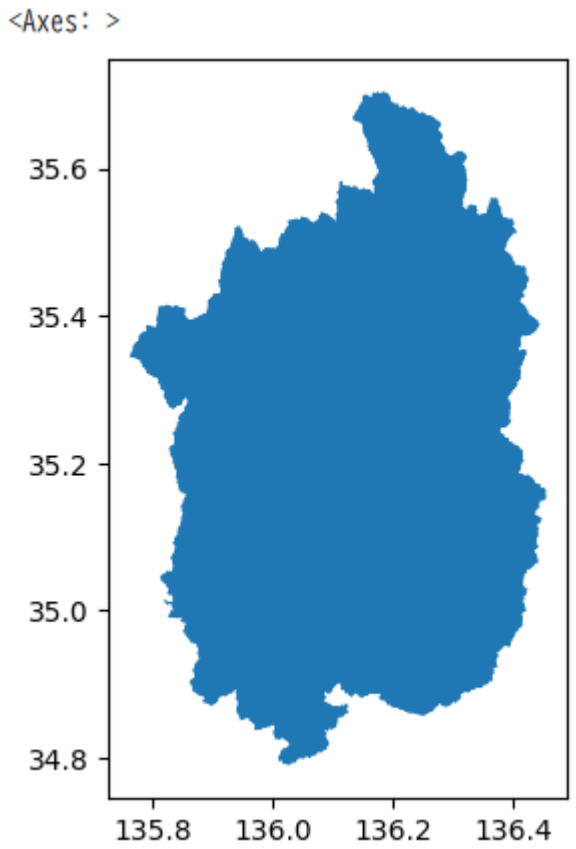
| | 0 |
|---|---|
| 0 | Polygon |
| 1 | Polygon |
| 2 | Polygon |
| 3 | Polygon |
| 4 | Polygon |

**dtype:** object

# GeoSeriesの処理

GeoDataFrameまたはGeoSeriesのplot()メソッドを実行すると、Matplotlibによる描画が行われる

```
geoseries.plot()
```



# GeoSeriesの処理

長さや面積に関する属性にアクセスすると、UserWarning例外が送出される
これは、地理的な座標を処理するには適切なCRSに投影する必要がある

```
geoseries.length.head(3)
```

```
/tmp/ipython-input-3314103464.py:1: UserWarning: Geometry is in a geographic CRS. Results from 'length' are likely
```

geoseries.length.head(3)

| | 0 |
|---|---|
| **0** | 4.172540 |
| **1** | 1.741525 |
| **2** | 0.896302 |

**dtype:** float64

# GeoSeriesの処理

to_crs()メソッドは、引数に渡したEPSGコードを元に指定した座標系に変換する。
これを行うことで、面積や距離などを算出できる

```
geoseries_transformed = geoseries.to_crs("EPSG:6674")
geoseries_transformed.length.head(3)
```

| | 0 |
|---|---|
| 0 | 421133.400718 |
| 1 | 176753.577131 |
| 2 | 89430.450606 |

**dtype:** float64

```
geoseries_transformed.area.head(3)
```

| | 0 |
|---|---|
| 0 | 4.016153e+09 |
| 1 | 4.644209e+08 |
| 2 | 1.968331e+08 |

**dtype:** float64

# 地理空間データの可視化

10-5-visualization.ipynb

# ポイントの可視化

- ポイント（Point）は、CSV形式など、表形式で提供されることもある
- CSV形式のファイルをpandasに読み込んで、Foliumを利用して可視化する
- データはG空間情報センターが公開する「伊丹市公衆無線LANアクセスポイントデータ（2022年4月より）(https://www.geospatial.jp/ckan/dataset/itami-free-wifi)」を利用する

```
import pandas as pd

itami_wifi = pd.read_csv("data/282073publicwirelesslan202204.csv")
itami_wifi.iloc[:3, 3:10]
```

| | 名称 | 名称_カナ | 名称_英語 | 住所 | 方書 | 緯度 | 経度 |
|---|---|---|---|---|---|---|---|
| 0 | 有岡城跡公園 | アリオカジョウセキコウエン | Arioka Castle Ruins | 兵庫県伊丹市伊丹1-16 | NaN | 34.780851 | 135.421042 |
| 1 | 市バス総合案内所 | シバスソウゴウアンナイジョ | City Bus Terminal | 兵庫県伊丹市西台1-1-24 | NaN | 34.780171 | 135.413086 |
| 2 | 市立伊丹ミュージアム | シリツイタミミュージアム | Itami City Museum | 兵庫県伊丹市宮ノ前2-5 | NaN | 34.781996 | 135.417263 |

G空間情報センター：https://front.geospatial.jp/

# ポイントの可視化

- 地図を描画するために、中心となる位置を算出する
- 下記コードで「緯度」列と「経度」列の平均値を算出する

```
itami_center = itami_wifi.loc[:, ["緯度", "経度"]].mean()
itami_center
```

| | 0 |
|---|---|
| **緯度** | 34.781261 |
| **経度** | 135.422682 |

**dtype:** float64

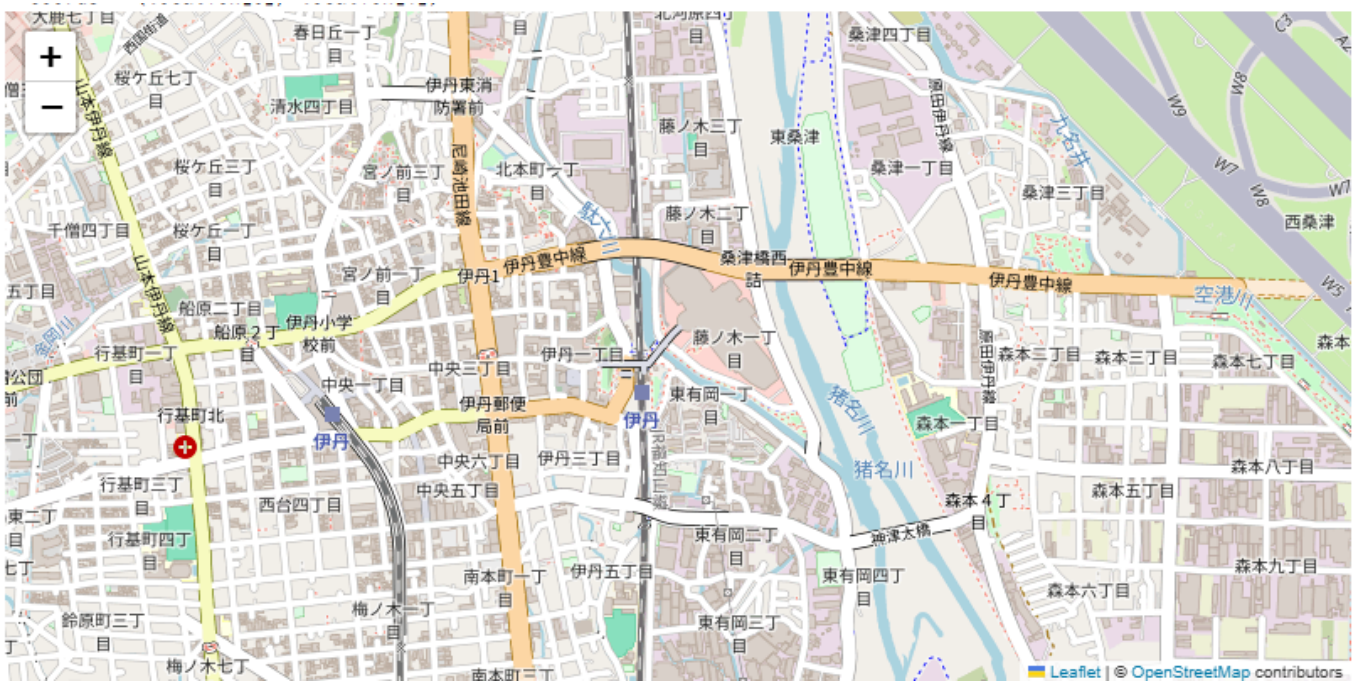
# ポイントの可視化

- Foliumは、Leaflet（leaflet.js）のPythonラッパーである
- Leafletは、地図を描画するための JavaScriptライブラリである
- Foliumを利用することで、地理データをWebアプリケーションや Jupyter Notebook 上にインタラクティブに操作できる地図を描画できる

# ポイントの可視化

Foliumを利用して前のコードで算出した位置情報を中心に、地図を描画している
Jupyter NotebookからはMapインスタンスをREPLに表示することで、地図が描画される。
Mapクラスの引数locationには、緯度と経度の値が含まれたlist-likeを渡す。
引数zoom_startは、描画時のズームレベルを指定する。引数widthとheightには、描画領域の幅と高さを指定する

```
import folium

itami_wifi_map = folium.Map(
    location=itami_center,
    zoom_start=15.2,
    width=800,
    height=400,
)
itami_wifi_map
```

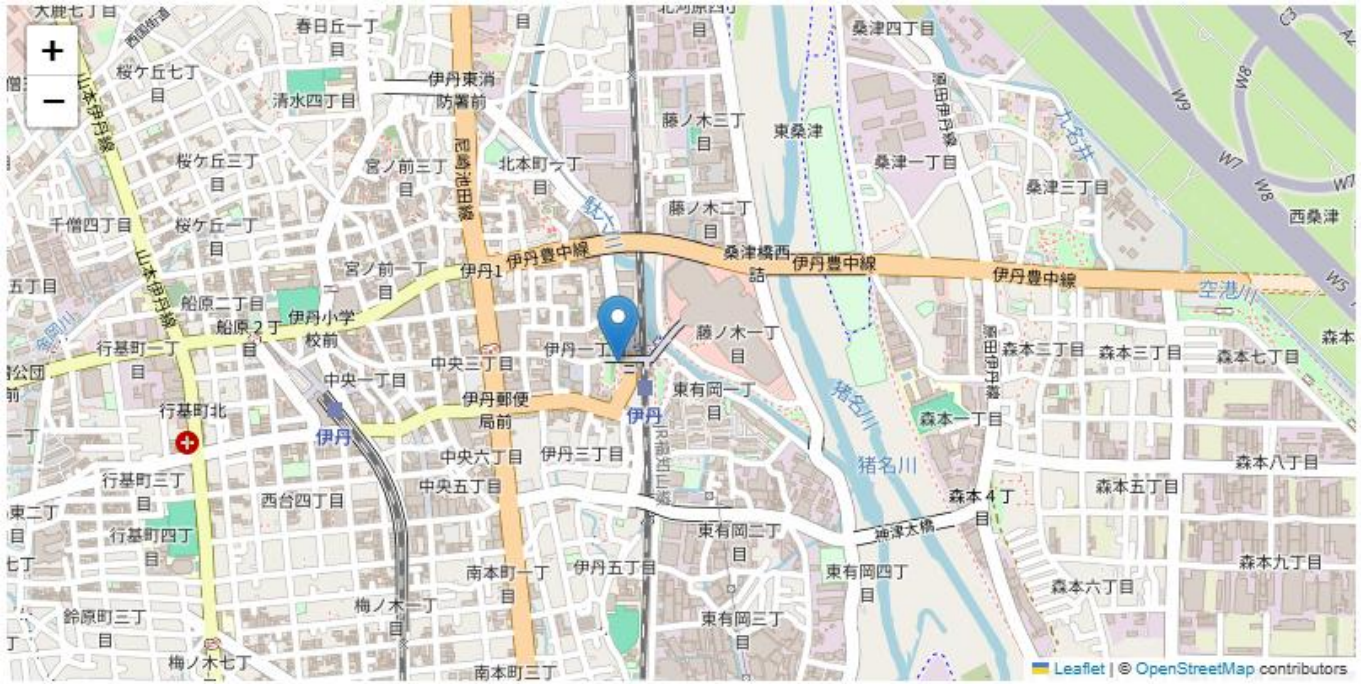



REPLとは？ REPL（Read-Evaluate-Print Loop）はPythonインタプリタと直接対話する方法

# ポイントの可視化

Markerクラスは、リーフレット型のマーカーを生成する
引数locationには、緯度と経度の値が 含まれたlist-likeを渡す。
Markerインスタンスからadd_to()メソッドを実行すると、引数に渡した Mapインスタンスにマーカーが追加される

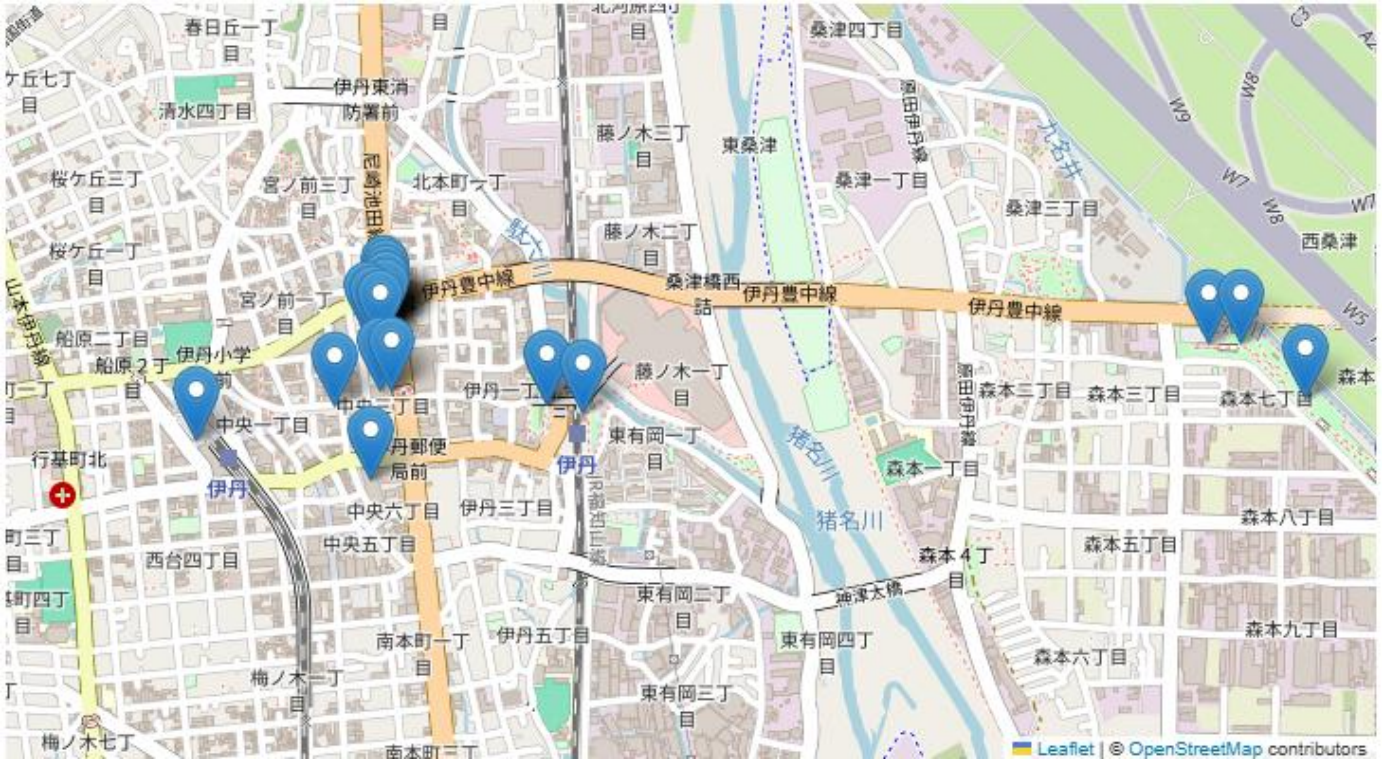
```
marker = folium.Marker(location=itami_wifi.loc[0, ["緯度", "経度"]])
marker.add_to(itami_wifi_map)
itami_wifi_map
```



# ポイントの可視化

DataFrameからapply()メソッドを実行して、すべての行の位置をマーカーに可視 化する

```
itami_wifi.loc[:, ["緯度", "経度"]].apply(
    lambda x: folium.Marker(location=x).add_to(itami_wifi_map),
    axis=1,
)
itami_wifi_map
```



# ヒートマップに可視化

GeoPandasに読み込んだポイントのデータをFoliumで地図に可視化します。データは、国土交通省「国土数値情報ダウンロードサービス」の滋賀県「地価公示データ（令和4年）」を利用する

```
import geopandas as gpd

land_price_gdf = (
    gpd.read_file(
        "https://nlftp.mlit.go.jp/ksj/gml/data/L01/L01-22/L01-22_25_GML.zip"
    )
    .groupby("L01_005")
    .get_group("2022")
)
land_price_gdf.loc[:3, ["L01_006", "L01_023", "geometry"]]
```

| | L01_006 | L01_023 | geometry |
|---|---|---|---|
| 0 | 61400 | 大津 | POINT (135.91198 35.10887) |
| 1 | 94200 | 大津 | POINT (135.91556 35.12295) |
| 2 | 41800 | 大津 | POINT (135.90057 35.09722) |
| 3 | 75400 | 大津 | POINT (135.87669 35.0748) |

国土数値情報ダウンロードサービス > 国土数値情報 > 地価公示データ https://nlftp.mlit.go.jp/ksj/gml/datalist/KsjTmplt-L01v3_1.html#prefecture24

# ヒートマップに可視化

GeoPandasに読み込んだポイントのデータをFoliumで地図に可視化します。データは、国土交通省「国土数値情報ダウンロードサービス」の滋賀県「地価公示データ（令和4年）」を利用する

```
import geopandas as gpd

land_price_gdf = (
    gpd.read_file(
        "https://nlftp.mlit.go.jp/ksj/gml/data/L01/L01-22/L01-22_25_GML.zip"
    )
    .groupby("L01_005")
    .get_group("2022")
)
land_price_gdf.loc[:3, ["L01_006", "L01_023", "geometry"]]
```

| | L01_006 | L01_023 | geometry |
|---|---|---|---|
| 0 | 61400 | 大津 | POINT (135.91198 35.10887) |
| 1 | 94200 | 大津 | POINT (135.91556 35.12295) |
| 2 | 41800 | 大津 | POINT (135.90057 35.09722) |
| 3 | 75400 | 大津 | POINT (135.87669 35.0748) |

「L01_006」列は、標準地の地価

国土数値情報ダウンロードサービス > 国土数値情報 > 地価公示データ https://nlftp.mlit.go.jp/ksj/gml/datalist/KsjTmplt-L01v3_1.html#prefecture24

# ヒートマップに可視化

データをヒートマップに可視化するため、「緯度, 経度, 地価」の3列となるnumpy.ndarray型に変換してする

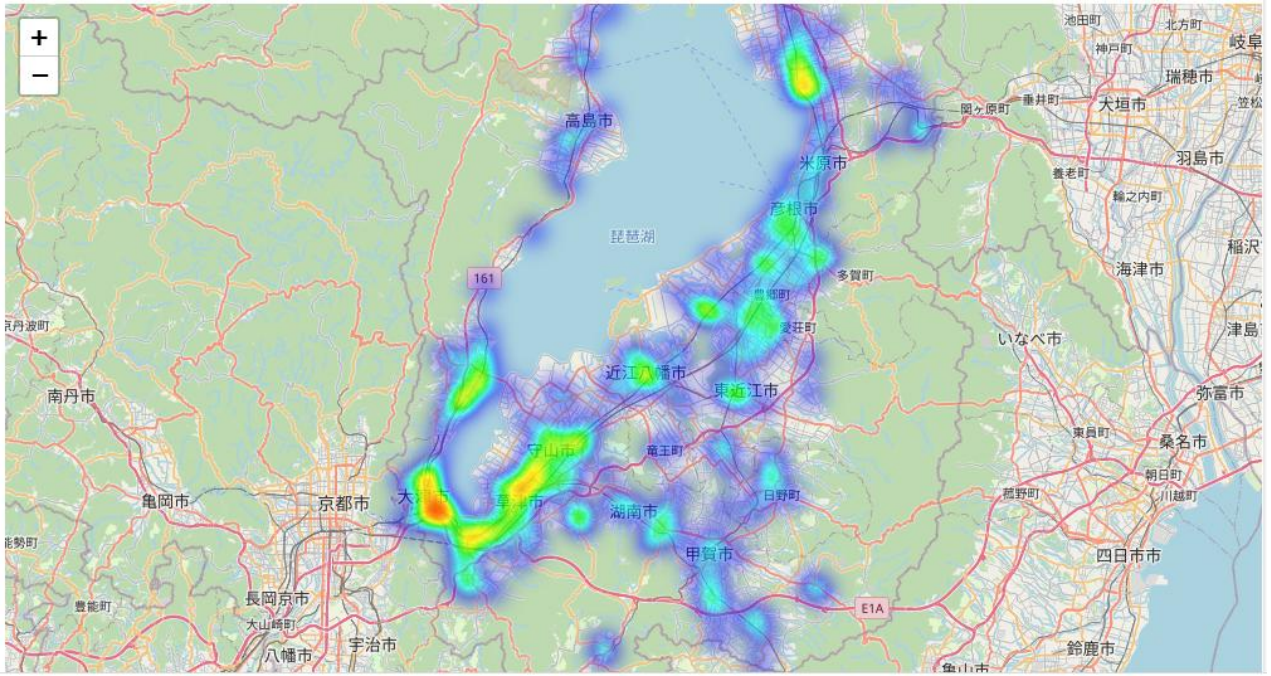
```
land_price = land_price_gdf.apply(
    lambda s: pd.Series(
        (
            s["geometry"].y,
            s["geometry"].x,
            s["L01_006"],
        )
    ),
    axis=1,
).values
land_price[:3]
```

```
array([[3.51088681e+01, 1.35911983e+02, 6.14000000e+04],
       [3.51229489e+01, 1.35915560e+02, 9.42000000e+04],
       [3.50972189e+01, 1.35900566e+02, 4.18000000e+04]])
```

# ヒートマップに可視化

滋賀県の地価をヒートマップで表現し、地図に描画している。plugins.HeatMapク ラスは、ヒートマップを生成する。引数dataには、上記のコードで生成したnumpy.ndarray型の land_priceを渡す。引数radiusには、ヒートマップの各点からの半径となる値を渡す

```
from folium.plugins import HeatMap

shiga_center = (
    land_price_gdf.loc[:, "geometry"].y.mean(),
    land_price_gdf.loc[:, "geometry"].x.mean(),
)
land_price_map = folium.Map(
    location=shiga_center,
    zoom_start=10,
)
HeatMap(
    data=land_price,
    radius=12,
).add_to(land_price_map)
land_price_map
```



# ラインの可視化

GeoPandasに読み込んだラインのデータをFoliumで地図に可視化する。
データは、国土交通省「国土数値情報ダウンロードサービス」の「鉄道データ（令和3年）」を利用する

GeoDataFrameから.cxアクセサを利用して滋賀県の地価データ と同じ範囲を抽出する

```
minx, miny = land_price_gdf.loc[:, "geometry"].bounds.min()[
    ["minx", "miny"]
]
maxx, maxy = land_price_gdf.loc[:, "geometry"].bounds.max()[
    ["maxx", "maxy"]
]

rail_gdf = gpd.read_file(
  #"https://nlftp.mlit.go.jp/ksj/gml/data/N02/N02-21/N02-21_GML.zip"
  “data/N02-21_RailroadSection.geojson” <- ローカルファイルを読み込む
).cx[minx:maxx, miny:maxy]
rail_gdf.head()
```

| | N02_001 | N02_002 | N02_003 | N02_004 | geometry |
|---|---|---|---|---|---|
| 1381 | 21 | 4 | 京津線 | 京阪電気鉄道 | LINESTRING (135.86366 35.01145, 135.86436 35.0... |
| 1382 | 21 | 4 | 京津線 | 京阪電気鉄道 | LINESTRING (135.86366 35.01145, 135.8634 35.01... |
| 1385 | 21 | 4 | 京津線 | 京阪電気鉄道 | LINESTRING (135.83788 34.99126, 135.83712 34.9... |
| 1387 | 21 | 4 | 京津線 | 京阪電気鉄道 | LINESTRING (135.83712 34.99145, 135.83601 34.9... |
| 1388 | 21 | 4 | 京津線 | 京阪電気鉄道 | LINESTRING (135.85273 34.99438, 135.85251 34.9... |

鉄道データ（令和3年）：https://nlftp.mlit.go.jp/ksj/gml/datalist/KsjTmplt-N02-v3_0.html

# ラインの可視化

「N02_003」列が「京津線」の行から「geometry」列を抽出し、中心となる座標を算出する。

LineStringのcentroid.yとcentroid.x属性から緯度と経度の中心を取得する。

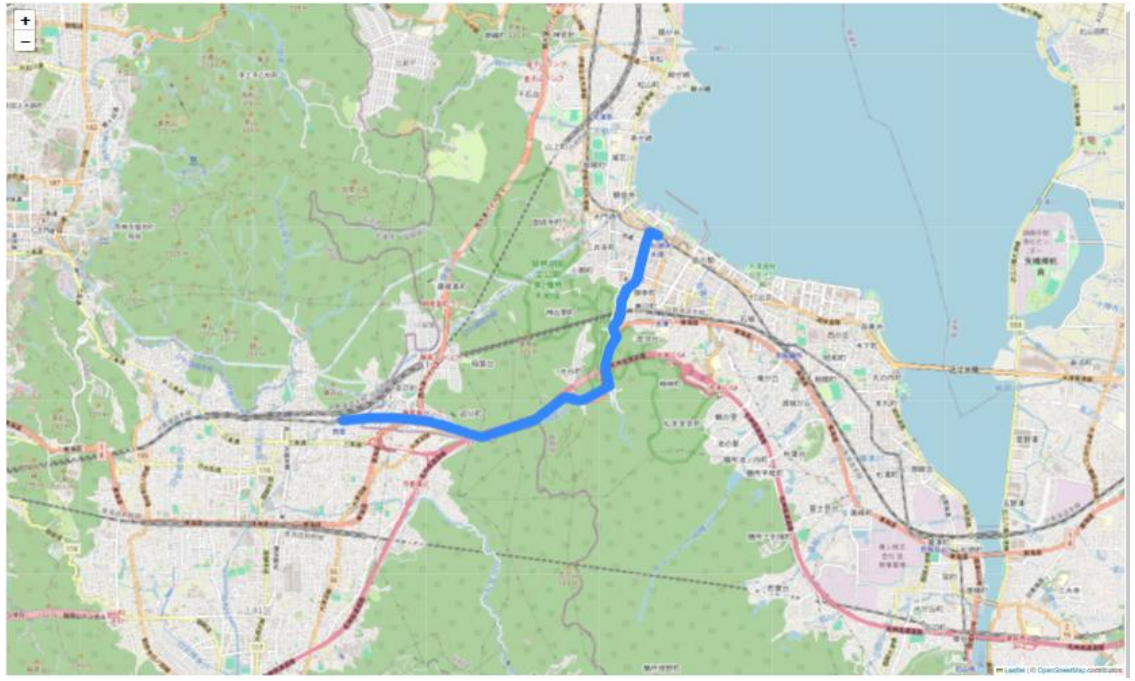
```
# 京津線のデータを抽出
keishinsen = rail_gdf.groupby("N02_003")["geometry"].get_group("京津線")

# 中心となる座標
keishinsen_center = (
    keishinsen.map(lambda p: p.centroid.y).mean(),
    keishinsen.map(lambda p: p.centroid.x).mean(),
)
keishinsen_center
```

(np.float64(35.00045506603767), np.float64(135.8527293845286))

# ラインの可視化

LineStringの座標から地図上に線を描画する。

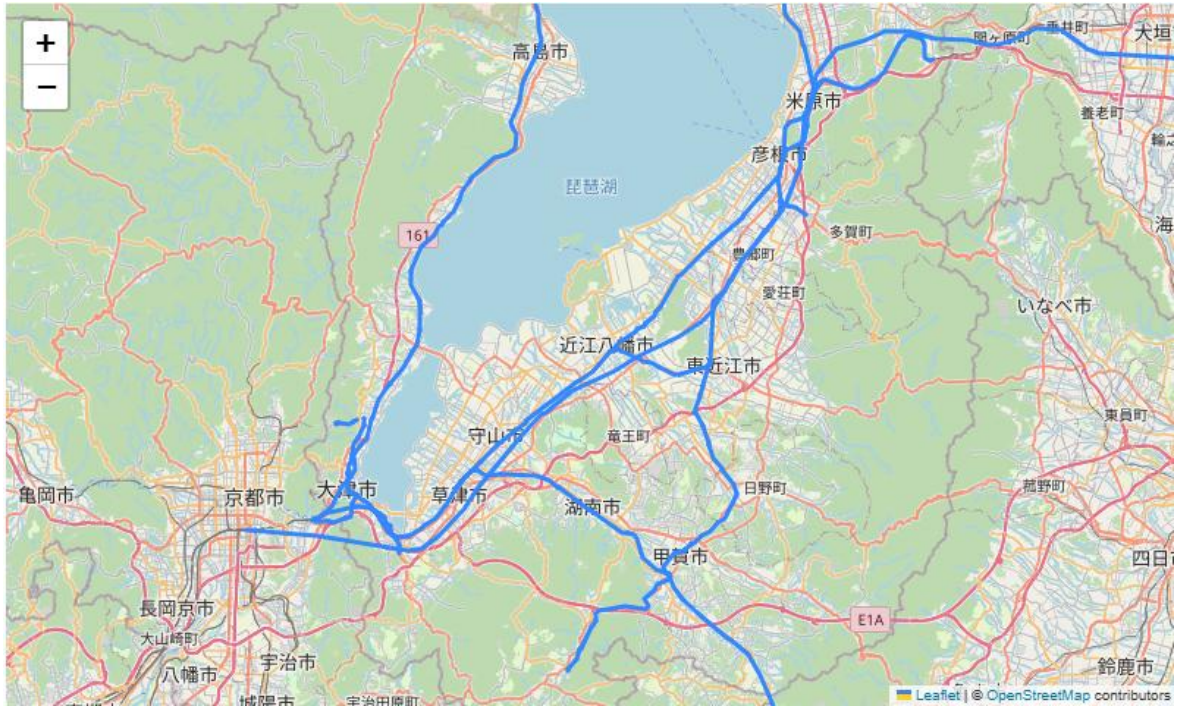
```
import numpy as np


keishinsen_map = folium.Map(
    location=keishinsen_center,
    zoom_start=14,
)

keishinsen.apply(
    lambda s: folium.PolyLine(
        # 緯度, 経度の順番に変換、fliplr関数で順序を反転
        locations=np.fliplr(s.coords),
        weight=15  # 線の太さ
    ).add_to(keishinsen_map)
)
keishinsen_map
```



# ラインの可視化

features.GeoJsonクラスは、GeoJSON形式のデータを描画する。features.GeoJsonクラスの引 数には、GeoJSON形式のファイルのほか、GeometricオブジェクトやGeoDataFrameなどが利用できる。　次のコードでは、GeoDataFrameを地図に描画している

```
rail_map = folium.Map(
    location=shiga_center,
    zoom_start=10,
)
folium.features.GeoJson(rail_gdf).add_to(rail_map
)
rail_map
```



# ポリゴンの可視化

GeoPandasに読み込んだポリゴンのデータをFoliumを使って地図に可視化する。

データ タは、国土交通省「国土数値情報ダウンロードサービス」の滋賀県「人口集中地区データ（平成27 年）」を利用する

国土数値情報ダウンロードサービス > 国土数値情報 > 人口集中地区データ
https://nlftp.mlit.go.jp/ksj/gml/datalist/KsjTmplt-A16-v2_3.html#prefecture24

# ポリゴンの可視化

## 滋賀県「人口集中地区データ（平成27 年）」を読み込む

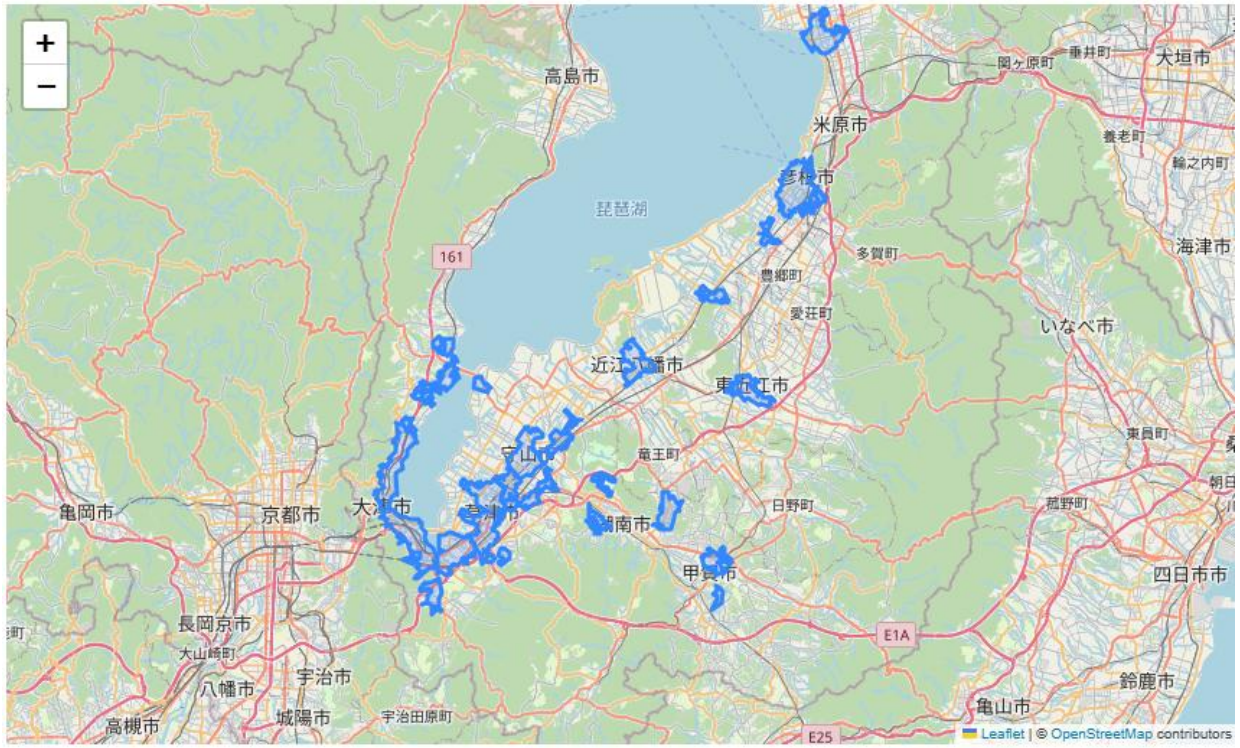
```
did_gdf = gpd.read_file("./data/A16-15_25_DID.geojson")
did_gdf.head()
```

| | DIDid | 行政コード | 市町村名称 | DID符号 | 人口 | 面積 | 前回人口 | 前回面積 | 人口割合 | 面積割合 | 調査年度 | geometry |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2520101 | 25201 | 大津市 | 1 | 137645 | 20.56 | 136520 | 20.48 | 40.4 | 4.4 | 2015 | POLYGON ((135.8691 35.08068, 135.86876 35.0801... |
| 1 | 2520101 | 25201 | 大津市 | 1 | 137645 | 20.56 | 136520 | 20.48 | 40.4 | 4.4 | 2015 | POLYGON ((135.8691 35.08068, 135.87006 35.0809... |
| 2 | 2520102 | 25201 | 大津市 | 2 | 60475 | 8.01 | 56260 | 7.33 | 17.7 | 1.7 | 2015 | POLYGON ((135.91999 34.99576, 135.92019 34.996... |
| 3 | 2520103 | 25201 | 大津市 | 3 | 19646 | 2.95 | 18661 | 2.77 | 5.8 | 0.6 | 2015 | POLYGON ((135.90632 35.10916, 135.9057 35.1092... |
| 4 | 2520103 | 25201 | 大津市 | 3 | 19646 | 2.95 | 18661 | 2.77 | 5.8 | 0.6 | 2015 | POLYGON ((135.90632 35.10916, 135.90682 35.108... |

# ポリゴンの可視化

鉄道データと同様に「geometry」列を地図に描画する

```
did_map = folium.Map(
    location=shiga_center,
    zoom_start=10,
)
folium.GeoJson(did_gdf).add_to(did_map)
did_map
```



# ポリゴンの可視化

「人口」列を値としたコロプレスマップを描画する

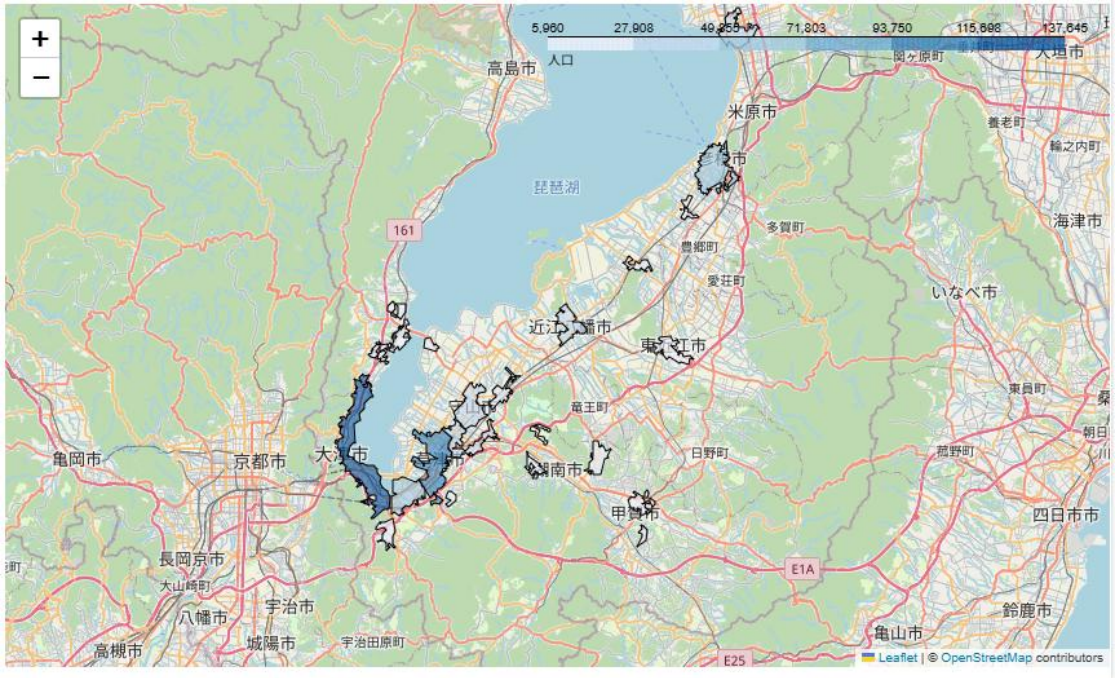
コロプレスマップ（階級 区分図）は、区域単位ごとの値を色調で表現した地図である

features.Choroplethクラスは、コロプ レスマップを生成します。コードで使用したfeatures.Choroplethクラスの引数は表のとおり です。

表 10-4　features.Choroplethクラスの引数

| 引数 | 説明 |
|---|---|
| geo_data | GeoJSON形式のデータ（GeoJSON形式のファイル、GeoDataFrameなど） |
| data | GeoJSON形式のデータにバインドするデータ |
| key_on | データをバインドするための引数dataのデータ内の変数、featureで始まるJavaScriptオブジェクト表記に従った文字列 |
| columns | 引数dataのデータがDataFrameであった場合の列名、「キー列，値列」の順番としたリスト |
| legend_name | 凡例のタイトル |

```
did_choroplethmap = folium.Map(
    location=shiga_center,
    zoom_start=10,
)
folium.features.Choropleth(
    geo_data=did_gdf,
    data=did_gdf,
    key_on="feature.properties.DIDid",
    columns=["DIDid", "人口"],
    legend_name="人口",
).add_to(did_choroplethmap)
did_choroplethmap
```



# ジオコーディング

10-column-geocoding.ipynb

# ジオコーディング

```
import json
from io import StringIO
from urllib import parse, request

import folium
import geopandas as gpd

place = "区役所"
url = parse.urlunparse(
    (
        "https",
        "msearch.gsi.go.jp",
        "/address-search/AddressSearch",
        None,
        parse.urlencode({"q": place}),
        None,
    )
)
res = request.urlopen(url)
geojson = StringIO(
    json.dumps(
        {
            "type": "FeatureCollection",
            "features": json.loads(res.read().decode()),
        }
    )
)
gdf = gpd.read_file(geojson)
center = (
    gdf.loc[:, "geometry"].y.mean(),
    gdf.loc[:, "geometry"].x.mean(),
)
m = folium.Map(location=center, zoom_start=7)
gdf.apply(
    lambda s: folium.Marker(
        location=[
            s["geometry"].y,
            s["geometry"].x,
        ]
    ).add_to(m),
    axis=1,
)
m
```

- ジオコーディングは、住所や地名から緯度経度などの地理座標を付与する処理である
- 明確な地理座標が存在しないデータは、ジオコーディングを行い、地理空間データを処理する場合がある
- ジオコーディングの例として、国土地理院の地名検索サービスから「区役所」というキーワードで緯度 経度を取得し、Folium を利用して地図に描画する