

# Dados Espaciais

## Conteúdo

- Fundamentos de estruturas de dados geográficos
- Híbridos
- Conclusão
- Perguntas

Este capítulo fundamenta as ideias discutidas nos dois capítulos anteriores em um contexto prático. Consideramos como as estruturas de dados e os modelos de dados que elas representam são implementados em Python. Também abordamos como interagir com essas estruturas de dados. Isso acontecerá junto com o código usado para manipular os dados em um único notebook de laboratório computacional. Isso, então, une os dois conceitos de ciência aberta e pensamento geográfico.

Além disso, passaremos a maior parte do capítulo discutindo como o Python representa os dados *uma vez lidos* de um arquivo ou banco de dados, em vez de focar em formatos de *arquivo específicos usados para armazenar dados*. Isso ocorre porque as bibliotecas que usamos lerão qualquer formato em uma das poucas estruturas de dados canônicas que discutimos no Capítulo 1. Adotamos essa abordagem porque essas estruturas de dados são o que interagimos durante nossa análise de dados: elas são nossa interface com os dados . Os formatos de arquivo, embora úteis, são secundários a esse propósito. De fato, parte do benefício do Python (e outras linguagens de computação) é *a abstração*: as complexidades, particularidades e peculiaridades associadas a cada formato de arquivo são removidas, pois o Python representa todos os dados de algumas maneiras padrão, independentemente da proveniência. Aproveitamos ao máximo esse recurso aqui.

Dividimos o capítulo em duas partes principais. A primeira parte examina cada uma das três principais estruturas de dados revisadas no Capítulo 1 ( *Pensamento Geográfico* ): tabelas geográficas, superfícies e gráficos espaciais. Em segundo lugar, exploramos combinações de diferentes estruturas de dados que se afastam das correspondências tradicionais de modelo/estrutura de dados discutidas no Capítulo 2. Cobrimos como um dado em uma estrutura pode ser efetivamente transferido para outra, mas também discutimos por que isso pode (ou não) ser uma boa ideia em alguns casos. Uma nota final antes de nos aprofundarmos no conteúdo deste livro é necessária: este não é um relato abrangente de *tudo*isso é possível com cada uma das estruturas de dados que apresentamos. Em vez disso, você pode pensar nisso como uma prévia da qual desenvolveremos ao longo do livro para mostrar muito do que é possível com o Python.

```
import pandas
import osmnx
import geopandas
import rioxarray
import xarray
import datashader
import contextily as cx
from shapely import geometry
import matplotlib.pyplot as plt
```

## Fundamentos das estruturas de dados geográficos

Conforme descrito no Capítulo 1, existem algumas estruturas de dados principais que são usadas na ciência de dados geográficos: tabelas geográficas (que geralmente correspondem a um modelo de dados de objeto), rasters ou superfícies (que geralmente correspondem a um modelo de dados de campo), e redes espaciais (que geralmente são combinadas com um modelo de dados de gráfico). Discutiremos estes por sua vez ao longo desta seção.

### Tabelas Geográficas

Os objetos geográficos geralmente correspondem ao que chamamos de *tabela geográfica* . As tabelas geográficas podem ser consideradas como uma guia em uma planilha onde uma das colunas registra informações geométricas. Essa estrutura de dados representa um único objeto geográfico como uma linha de uma tabela; cada coluna da tabela registra informações sobre o objeto, seus atributos ou características, como veremos a seguir. Normalmente, há uma coluna especial nesta tabela que registra a *geometria* do objeto. Os sistemas de computador que usam essa estrutura de dados destinam-se a adicionar geografia a um banco de *dados relacional*, como PostgreSQL (através de sua extensão PostGIS) ou sqlite (através de sua extensão espacialite). Além disso, no entanto, muitas linguagens de ciência de dados (como R, Julia e Python) têm pacotes que também adotam essa estrutura de dados (como `sf`, `GeoTables.jl` e `geopandas`), e ela está rapidamente se tornando a principal estrutura de dados para sistemas baseados em objetos. dados geográficos.

Antes de prosseguir, porém, é útil mencionar um rápido esclarecimento sobre a terminologia. Ao longo deste livro, independentemente da estrutura de dados utilizada, nos referiremos a uma medição sobre uma observação como uma *característica* . Isso é consistente com outros trabalhos em ciência de dados e aprendizado de máquina. Então, um conjunto de medições é uma *amostra* . Para tabelas, isso significa que um recurso é uma coluna e uma amostra é uma linha. Historicamente, porém, os cientistas da informação geográfica têm usado a palavra “recurso” para significar uma observação individual, uma vez que um “recurso” em cartografia é uma entidade em um mapa e “atributo” para descrever as características dessa observação. Em outros lugares, um recurso pode ser chamado de “variável” e uma amostra pode ser chamada de “registro”. Assim, uma terminologia consistente é importante: para este livro, um *recurso* é um traço medido pertencente a uma observação (uma coluna), e uma *amostra* é um conjunto de medições (uma linha).

Para entender a estrutura das tabelas geográficas, será útil ler o conjunto de `countries_clean.gpkg` dados incluído neste livro que descreve os países do mundo. Para ler esses dados, podemos usar o `read_file()` método em `geopandas`: [1](#).

```
gt_polygons = geopandas.read_file(
    "../data/countries/countries_clean.gpkg"
)
```

E podemos examinar o topo da tabela com o `.head()` método:

```
gt_polygons.head()
```

	ADMINISTRADOR	geometria
0	Indonésia	MULTIPOLÍGONO (((13102705.696 463877.598, 13102...
1	Malásia	MULTIPOLÍGONO (((13102705.696 463877.598, 13101...
2	Chile	MULTIPOLÍGONO (((-7737827.685 -1979875.500, -77...
3	Bolívia	POLÍGONO ((-7737827.685 -1979875.500, -7737828....
4	Peru	MULTIPOLÍGONO (((-7737827.685 -1979875.500, -77...

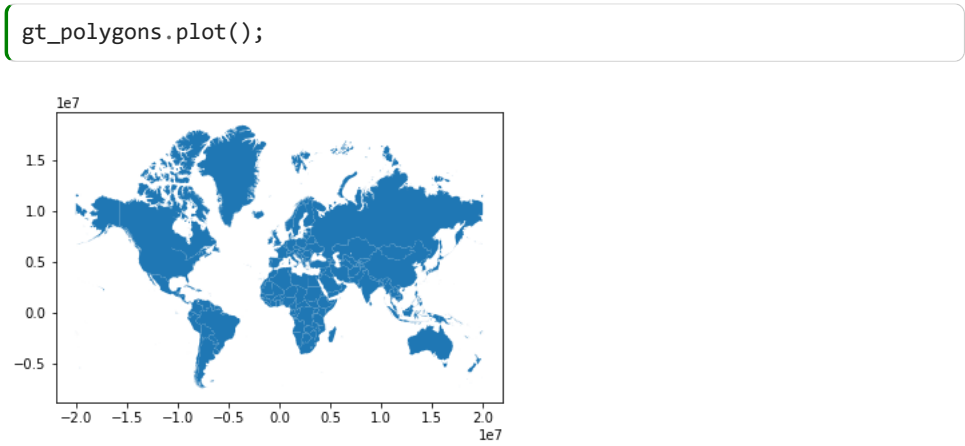
Cada linha desta tabela é um único país. Cada país tem apenas duas características: o nome administrativo do país e a geometria da fronteira do país. O nome do país é codificado na `ADMIN` coluna usando o `str` tipo Python, que é usado para armazenar dados baseados em texto. A geometria da fronteira do país é armazenada na `geometry` coluna e codificada usando uma classe especial em Python que é usada para representar objetos geométricos. Como em outras estruturas de dados baseadas em tabelas em Python, cada linha e coluna tem um índice que as identifica exclusivamente e é renderizado em negrito no lado esquerdo da tabela. Essa tabela geográfica é uma instância do `geopandas.GeoDataFrame` objeto, usada em todo o ecossistema do Python para representar dados geográficos.

As tabelas geográficas armazenam informações geográficas como uma coluna adicional. Mas, como essa informação é codificada? Para ver, podemos verificar o tipo do objeto na primeira linha:

```
type(gt_polygons.geometry[0])

shapely.geometry.multipolygon.MultiPolygon
```

Em `geopandas` (assim como em outros pacotes que representam dados geográficos), a `geometry` coluna possui características especiais que uma coluna “normal”, como `ADMIN`, não possui. Por exemplo, quando plotamos o dataframe, a `geometry` coluna é usada como a forma principal a ser usada na plotagem, conforme mostrado na Figura 1.



A alteração da representação geométrica de uma amostra deve ser feita com cuidado: como a `geometry` coluna é especial, existem funções especiais para ajustar a geometria. Por exemplo, se quisermos representar cada país usando seu *centróide*, um ponto no meio da forma, devemos ter o cuidado de garantir que uma nova coluna de geometria seja definida corretamente usando o `set_geometry()` método. Isso pode ser útil quando você deseja trabalhar com duas representações geométricas diferentes da mesma amostra subjacente.

Façamos um mapa da fronteira e do centróide de um país. Primeiro, para calcular o centróide, podemos usar a `gt_polygons.geometry.centroid` propriedade. Isso nos dá o ponto que minimiza a distância média de todos os outros pontos no limite da forma. Armazenando isso de volta em uma coluna, chamada `centroid`:

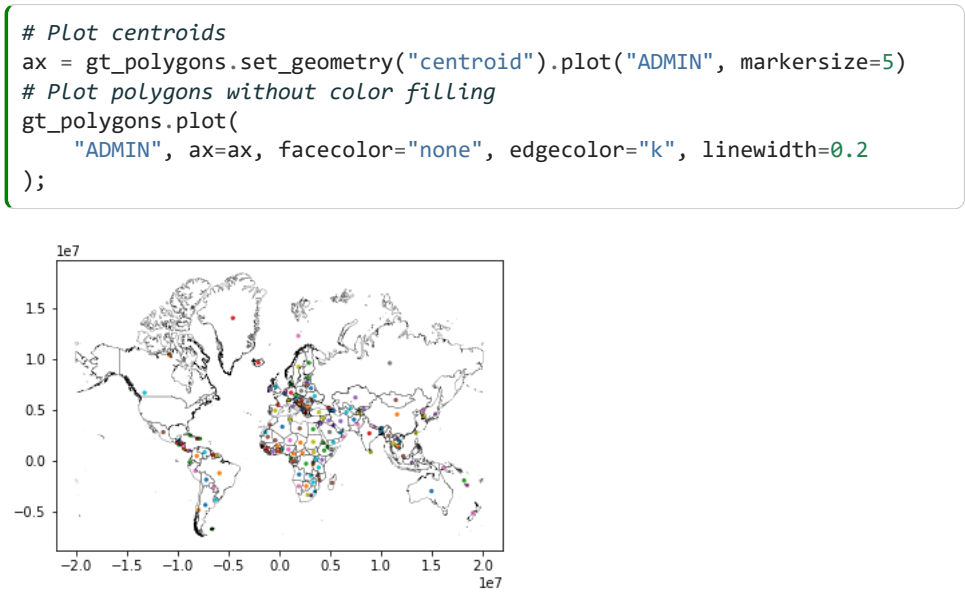
```
gt_polygons["centroid"] = gt_polygons.geometry.centroid
```

Agora temos um recurso adicional:

```
gt_polygons.head()
```

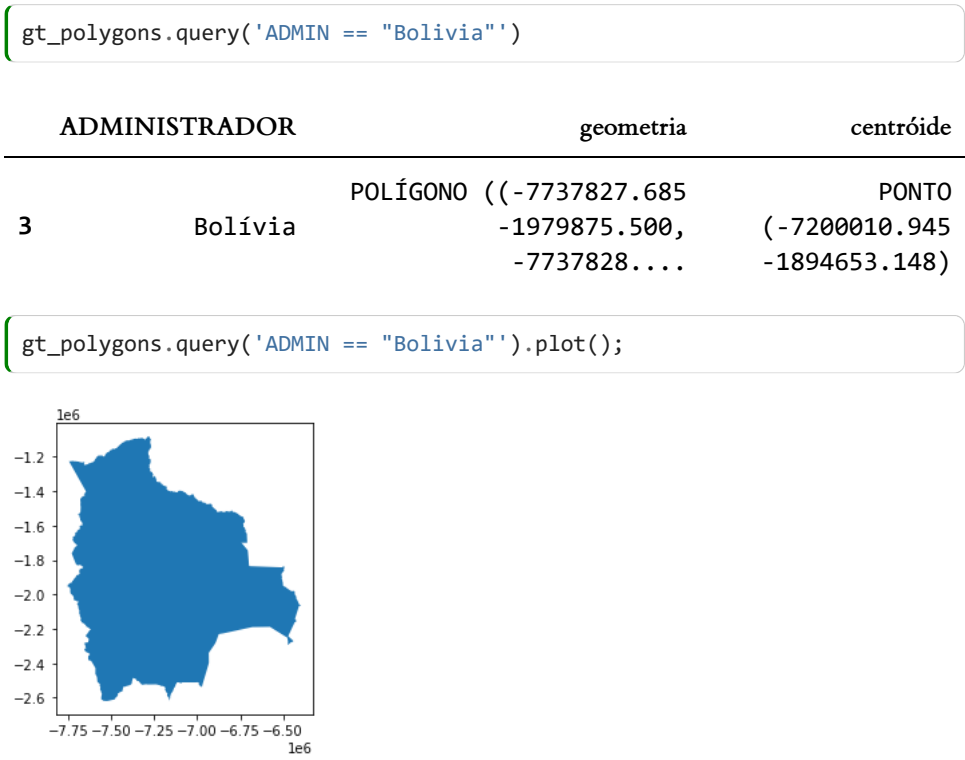
ADMINISTRADOR		geometria	centróide
0	Indonésia	MULTIPOLÍGONO	PONTO
		((((13102705.696 463877.598, 13102...	(13055431.810 -248921.141)
1	Malásia	MULTIPOLÍGONO	PONTO
		((((13102705.696 463877.598, 13101...	(12211696.493 422897.505)
2	Chile	MULTIPOLÍGONO	PONTO
		(((-7737827.685 -1979875.500, -77...	(-7959811.948 -4915458.802)
3	Bolívia	POLÍGONO ((-7737827.685 -1979875.500, -7737828....	PONTO
			(-7200010.945 -1894653.148)
4	Peru	MULTIPOLÍGONO	PONTO
		(((-7737827.685 -1979875.500, -77...	(-8277554.831 -1032942.536)

Apesar de `centroid` ser uma geometria (você pode dizer porque cada célula começa com `POINT`), ela não está definida como a geometria ativa para nossa tabela. Podemos mudar para a `centroid` coluna usando o `set_geometry()` método. Por fim, podemos plotar o centróide e o limite de cada país após alternar a coluna de geometria com `set_geometry()`:



Observe novamente como podemos criar um mapa chamando `.plot()` um arquivo `GeoDataFrame`. Podemos colorir tematicamente cada recurso com base em uma coluna, passando o nome dessa coluna para o método plot (como fazemos `ADMIN` neste caso) e que a geometria atual seja usada.

Assim, como já deve estar claro, praticamente qualquer tipo de objeto geográfico pode ser representado em uma (ou mais) coluna(s) geométrica(s). Pensar no número de diferentes tipos de formas ou geometrias que alguém poderia usar rapidamente confunde a mente. Felizmente, o Open Geospatial Consortium (OGC) definiu um conjunto de tipos “abstratos” que podem ser usados para definir qualquer tipo de geometria. Essa especificação, codificada na ISO 19125-1 – a especificação “simple features” – define as relações formais entre esses tipos: a `Point` é uma localização de dimensão zero com uma coordenada xey; a `LineString` é um caminho composto por um conjunto de mais de um `Point`, e a `Polygon` é uma superfície que possui pelo menos um `LineString` que começa e termina com a mesma coordenada. Todos esses tipos *também* têm `Multi` variantes que indicam uma coleção de várias geometrias do mesmo tipo. Assim, por exemplo, a Bolívia é representada como um único polígono:

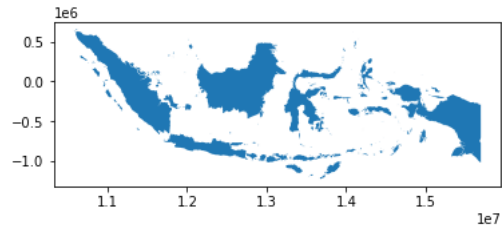


enquanto a Indonésia `MultiPolygon` contém muitos `Polygons` para cada ilha individual no país:



ADMINISTRADOR		geometria	centróide
0	Indonésia	MULTIPOLÍGONO	PONTO
		(( (13102705.696 463877.598, 13102...	(13055431.810 -248921.141)

```
gt_polygons.query('ADMIN == "Indonesia").plot();
```



Em muitos casos, as tabelas geográficas terão geometrias de um único tipo; registros serão todos ou **Point** , **LineString** por exemplo. No entanto, não há nenhuma exigência formal de que uma *tabela geográfica* tenha geometrias todas do mesmo tipo.

Ao longo deste livro, usaremos tabelas geográficas extensivamente, armazenando polígonos, mas também pontos e linhas. Exploraremos linhas um pouco mais na segunda parte deste capítulo, mas, por enquanto, vamos parar nos pontos por um segundo. Como mencionado acima, esses são os tipos de recursos mais simples, pois não possuem nenhuma dimensão, apenas um par de coordenadas anexadas a eles. Isso significa que às vezes os pontos podem ser armazenados em uma tabela não geográfica, simplesmente usando uma coluna para cada coordenada. Encontramos um exemplo disso no conjunto de dados de Tóquio que usaremos mais tarde. Os dados são armazenados como uma tabela de valores separados por vírgulas ou **.csv**:

```
gt_points = pandas.read_csv("../data/tokyo/tokyo_clean.csv")
```

Como a vemos com **pandas**, a tabela é carregada como **DataFrame**, sem dimensão espacial explícita:

```
type(gt_points)
```

```
pandas.core.frame.DataFrame
```

Se inspecionarmos a tabela, descobrimos que não há uma **geometry** coluna:

```
gt_points.head()
```

	ID do usuário	longitude	latitude	date_taken	foto/video_page_url	x	
0	10727420@N00	139.700499	35.674000	2010-04-09 17:26:25.0	http://www.flickr.com/photos/10727420@N00/4545...	1.555139e+07	↗
1	8819274@N04	139.766521	35.709095	2007-02-10 16:08:40.0	http://www.flickr.com/photos/8819274@N04/26503...	1.555874e+07	↗
2	62068690@N00	139.765632	35.694482	21/12/2008 15:45:31.0	http://www.flickr.com/photos/62068690@N00/3125...	1.555864e+07	↗
3	49503094041@N01	139.784391	35.548589	2011-11-11 05:48:54.0	http://www.flickr.com/photos/49503094041@N01/6...	1.556073e+07	↗
4	40443199@N00	139.768753	35.671521	2006-04-06 16:42:49.0	http://www.flickr.com/photos/40443199@N00/2482...	1.555899e+07	↗

Muitos conjuntos de dados de pontos são fornecidos neste formato. Para aproveitá-los ao máximo, é conveniente convertê-los em **GeoDataFrame** tabelas. Há duas etapas envolvidas neste processo. Primeiro, transformamos as coordenadas brutas em geometrias:

```
pt_geoms = geopandas.points_from_xy(  
    x=gt_points["longitude"],  
    y=gt_points["latitude"],  
    # x,y are Earth Longitude & Latitude  
    crs="EPSG:4326",  
)
```

Em segundo lugar, criamos um **GeoDataFrame** objeto usando estas geometrias:

```
gt_points = geopandas.GeoDataFrame(gt_points, geometry=pt_geoms)
```

E agora **gt\_points** se parece exatamente com o dos países que vimos antes, com a diferença que a **geometry** coluna armazena **POINT**s geometrias:

```
gt_points.head()
```

17/11/2022 18:53

Dados Espaciais — Ciência de Dados Geográficos com Python

	ID do usuário	longitude	latitude	date_taken	foto/video_page_url	x
0	10727420@N00	139.700499	35.674000	2010-04-09 17:26:25.0	http://www.flickr.com/photos/10727420@N00/4545...	1.555139e+07
1	8819274@N04	139.766521	35.709095	2007-02-10 16:08:40.0	http://www.flickr.com/photos/8819274@N04/26503...	1.555874e+07
2	62068690@N00	139.765632	35.694482	21/12/2008 15:45:31.0	http://www.flickr.com/photos/62068690@N00/3125...	1.555864e+07
3	49503094041@N01	139.784391	35.548589	2011-11-11 05:48:54.0	http://www.flickr.com/photos/49503094041@N01/6...	1.556073e+07
4	40443199@N00	139.768753	35.671521	2006-04-06 16:42:49.0	http://www.flickr.com/photos/40443199@N00/2482...	1.555899e+07

Superfícies

As superfícies são usadas para registrar dados de um modelo de dados de campo. Em teoria, um campo é uma superfície contínua e, portanto, possui um número infinito de locais nos quais pode ser medido. Na realidade, no entanto, os campos são medidos em uma amostra finita de locais que, para fornecer uma sensação de continuidade e melhor conformidade com o modelo de campo, são estruturados uniformemente no espaço. As superfícies, portanto, são representadas como grades onde cada célula contém uma amostra. Uma grade também pode ser considerada uma tabela com linhas e colunas, mas, como discutimos no capítulo anterior, ambas estão diretamente ligadas à localização geográfica. Isso contrasta fortemente com as tabelas geográficas, onde a geografia é confinada a uma única coluna.

Para explorar como Python representa superfícies, usaremos uma extração para a cidade brasileira de São Paulo de um conjunto de [dados de população global](#) . Este conjunto de dados registra contagens populacionais em células das mesmas dimensões cobrindo uniformemente a superfície da Terra. Nosso extrato está disponível como um arquivo GeoTIF, uma variação do formato de imagem TIF que inclui informações geográficas. Podemos usar o `open_rasterio()` método do `xarray` pacote para ler no GeoTIF:

```
pop = xarray.open_rasterio("../data/ghsl/ghsl_sao_paulo.tif")
```

Isso lê os dados em um `DataArray` objeto:

```
type(pop)

xarray.core.dataarray.DataArray
```

`xarray` é um pacote para trabalhar com arrays rotulados multidimensionais. Vamos descompactar isso: podemos usar arrays não apenas de duas dimensões como em uma tabela com linhas e colunas, mas com um número arbitrário delas; cada uma dessas dimensões é “rastreada” por um índice que torna fácil e eficiente a manipulação. Em `xarray`, esses índices são chamados de coordenadas e podem ser recuperados de nosso `DataArray` por meio do `coords` atributo:

```
pop.coords

Coordinates:
  * band      (band) int64 1
  * y         (y) float64 -2.822e+06 -2.822e+06 ... -2.926e+06 -2.926e+06
  * x         (x) float64 -4.482e+06 -4.482e+06 ... -4.365e+06 -4.365e+06
```

Interestingly, our surface has *three* dimensions: `x`, `y`, and `band`. The former two track the latitude and longitude that each cell in our population grid covers. The third one has a single value (1) and, in this context, it is not very useful. But it is easy to imagine contexts where a third dimension would be useful. For example, an optical color image may have three bands: red, blue, and green. More powerful sensors may pick up additional bands, such as near infrared (NIR) or even radio bands. Or, a surface measured over time, like the geocubes that we discussed in Chapter 2, will have bands for each point in time at which the field is measured. A geographic surface will thus have two dimensions recording the location of cells (`x` and `y`), and at least one `band` that records other dimensions pertaining to our data.

An `xarray.DataArray` object contains additional information about the values stored under the `attrs` attribute:

```
pop.attrs

{'transform': (250.0, 0.0, -4482000.0, 0.0, -250.0, -2822000.0),
 'crs': '+proj=moll +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs=True',
 'res': (250.0, 250.0),
 'is_tiled': 0,
 'nodatavals': (-200.0,),
 'scales': (1.0,),
 'offsets': (0.0,),
 'AREA_OR_POINT': 'Area',
 'grid_mapping': 'spatial_ref'}
```



In this case, we can see this includes information required to convert pixels in the array into locations on the Earth surface (e.g., `transform`, and `crs`), the spatial resolution (250 meters by 250 meters), and other metadata that allows us to better understand where the data comes from and how it is stored.

Thus, our `dataArray` has three dimensions:

```
pop.shape

(1, 416, 468)
```

A common operation will be to reduce this to only the two geographic ones. We can do this with the `sel` operator, which allows us to select data by the value of their coordinates:

```
pop.sel(band=1)

xarray.DataArray (y: 416, x: 468)
[194688 values with dtype=float32]
Coordinates:
  band      ()      int64  1
  y         (y) float64 -2.822e+06 ... -2.926e+06
  x         (x) float64 -4.482e+06 ... -4.365e+06
Attributes:
  transform : (250.0, 0.0, -4482000.0, 0.0, -250.0, -2822000.0)
  crs :       +proj=moll +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units
              =m +no_defs=True
  res :       (250.0, 250.0)
  is_tiled :   0
  nodatavals : (-200.0,)
  scales :     (1.0,)
  offsets :    (0.0,)
  AREA_O...   Area
  grid_mappi... spatial_ref
```

The resulting object is thus a two-dimensional array. Similar to geographic tables, we can quickly plot the values in our dataset:

```
pop.sel(band=1).plot();
```

This gives us a first overview of the distribution of population in the Sao Paulo region. However, if we inspect the map further, we can see that the map includes negative counts! How could this be? As it turns out, missing data are traditionally stored in surfaces not as a class of its own (e.g., `NaN`) but with an impossible value. If we return to the `attrs` printout above, we can see how the `nodatavals` attribute specifies missing data recorded with `-200`. With that in mind, we can use the `where()` method to select only values that are *not* `-200`:

```
pop.where(pop != -200).sel(band=1).plot(cmap="RdPu");
```

The colorbar now looks more sensible, and indicates *real* counts, rather than including the missing data placeholder values.

Spatial graphs

Spatial graphs store connections between objects through space. These connections may derive from geographical topology (e.g., contiguity), distance, or more sophisticated dimensions such as interaction flows (e.g., commuting, trade, communication). Compared to geographic tables and surfaces, spatial graphs are rather different. First, in most cases they do not record

measurements about a given phenomena, but instead focus on *connections*, on storing relationships between objects as they are facilitated (or impeded in their absence) by space. Second, because of this relational nature, the data are organized in a more unstructured fashion: while one sample may be connected to only one other sample, another one can display several links. This in stark contrast to geographic tables and surfaces, both of which have a clearly defined structure, shape and dimensionality in which data are organized.

These particularities translate into a different set of Python data structures. Unlike the previous data structures we have seen, there are quite a few data structures to represent spatial graphs, each optimized for different contexts. One such case is the use of spatial connections in statistical methods such as exploratory data analysis or regression. For this, the most common data structure are spatial weights matrices, to which we devote the next chapter.

In this chapter, we briefly review a different way of representing spatial graphs that is much closer to the mathematical concept of a graph. A graph is composed of *nodes* that are linked together by *edges*. In a spatial network, *nodes* may represent geographical places, and thus have a specific location; likewise, *edges* may represent geographical paths between these places. Networks require both *nodes* and *edges* to analyze their structure.

For illustration, we will rely on the `osmnx` library, which can query data from OpenStreetMap. For example, we extract the street-based graph of Yoyogi Park, near our earlier data from Tokyo:

```
graph = osmnx.graph_from_place("Yoyogi Park, Shibuya, Tokyo, Japan")
```

The code snippet above sends the query to the OpenStreetMap server to fetch the data. Note that the cell above *requires* internet connectivity to work. If you are working on the book *without* connectivity, a cached version of the graph is available on the data folder and can be read as:

```
graph = osmnx.load_graphml("../data/cache/yoyogi_park_graph.graphml")
```

Once the data is returned to `osmnx`, it gets processed into the `graph` Python representation:

```
type(graph)

networkx.classes.multidigraph.MultiDiGraph
```

We can have a quick inspection of the structure of the graph with the `plot_graph` method:

```
osmnx.plot_graph(graph);
```



The resultant `graph` object is actually a `MultiDiGraph` from `networkx`, a graph library written in Python. The graph here is stored as a collection of 106 nodes (street intersections):

```
len(graph.nodes)

106
```

and 287 edges (streets) that connect them:

```
len(graph.edges)

287
```

Each of these elements can be queried to obtain more information such as the location and ID of a node:

```
graph.nodes[1520546819]

{'y': 35.6711267, 'x': 139.6925951, 'street_count': 4}
```

The characteristics of an edge:

```
graph.edges[(1520546819, 3010293622, 0)]
```

```
{'osmid': 138670840,
'highway': 'footway',
'oneway': False,
'length': 59.113,
'geometry': <shapely.geometry.linestring.LineString at 0x7f3123456190>}
```

Or how the different components of the graph relate to each other. For example, what other nodes are directly connected to node 1520546819?

```
list(graph.adj[1520546819].keys())

[3010293622, 5764960322, 1913626649, 1520546959]
```

Thus, networks are easy to represent in Python, and are one of the three main data structures in geographic data science.

## Hybrids

We have just seen how geographic tables, surfaces, and networks map onto `GeoDataFrame`, `DataArray` and `Graph` objects in Python, respectively. These represent the conventional pairings that align data models to data structures with Python representations. However, while the conventional pairings are well-used, there are others in active use and many more to yet be developed. Interestingly, many new pairings are driven by new developments in technology, enabling approaches that were not possible in the past or creating situations (e.g., large datasets) that make the conventional approach limiting. Therefore, in this second section of the chapter, we step a bit “out of the box” to explore cases in which it may make sense to represent a dataset with a data structure that might not be the most obvious initial choice.

### Surfaces as tables

The first case we explore is treating surfaces as (geo-)tables. In this context, we shift from an approach where each dimension has a clear mapping to a spatial or temporal aspect of the dataset, to one where each sample, cell of the surface/cube is represented as a row in a table. This approach runs contrary to the general consensus that fields are best represented as surfaces or rasters because that allows us to index space and time “by default” based on the location of values within the data structure. Shifting to a tabular structure implies either losing that space-time reference, or having to build it manually with auxiliary objects (e.g., a spatial graph). In almost any case, operating on this format is less efficient than it *could* be if we had bespoke algorithms built around surface structures. Finally, from a more conceptual point of view, treating pixels as independent realizations of a process that we *know* is continuous can be computationally inefficient and statistically flawed.

This perspective however also involves important benefits. First, sometimes we *don’t* need location for our particular application. Maybe we are interested in calculating overall descriptive statistics; or maybe we need to run an analysis that is entirely atomic in the sense that it operates on each sample in isolation from all the other ones. Second, by “going tabular” we recast our specialized, spatial data into the most common data structure available, for which a large amount of commodity technology is built. This means many new tools can be used for analysis. So called “big data” technologies, such as distributed systems, are much more common, robust, and tested for tabular data than for spatial surfaces. *If* we can translate our spatial challenge into a tabular challenge, we can immediately plug in technology that is more optimized and, in some cases, reliable. Further, some analytic toolboxes common in (geographic) data science are entirely built around tabular structures. Machine learning packages such as `scikit-learn`, or some spatial analytics (such as most methods in the Pysal family of packages) are designed around this data structure. Converting our surfaces into tables thus allows us to plug into a much wider suite of (potentially) efficient tools and techniques.

We will see two ways of going from surfaces to tables: one converts every pixel into a table row, and another that aggregates pixels into predetermined polygons.

#### One pixel at a time

Technically, going from surface to table involves traversing from `xarray` to `pandas` objects. This is actually a well established bridge. To illustrate it with an example, let’s revisit the population counts in [Sao Paulo](#) used earlier. We can read the surface into a `DataArray` object with the `open_rasterio()` method:

```
surface = xarray.open_rasterio("../data/ghsl/ghsl_sao_paulo.tif")
```

Transferring to a table is as simple as calling the `DataArray`’s `to_series()` method:

```
t_surface = surface.to_series()
```

The resulting object is a `pandas.Series` object indexed on each of the dimensions of the original `DataArray`:

```
t_surface.head()

band  y          x
1    -2822125.0  -4481875.0  -200.0
      -4481625.0  -200.0
      -4481375.0  -200.0
      -4481125.0  -200.0
      -4480875.0  -200.0
dtype: float32
```



At this point, everything we know about **pandas** and tabular data applies! For example, it might be more convenient to express it as a **DataFrame**:

```
t_surface = t_surface.reset_index().rename(columns={0: "Value"})
```

With the power of a tabular library, some queries and filter operations become much easier. For example, finding cells with more than 1,000 people can be done with the usual `query()` method.[\[2\]](#)

```
t_surface.query("Value > 1000").info()

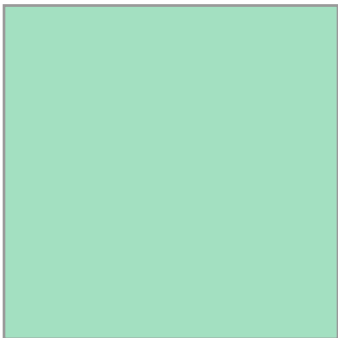
<class 'pandas.core.frame.DataFrame'>
Int64Index: 7734 entries, 3785 to 181296
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype  
---  -
 0   band    7734 non-null    int64   
 1   y        7734 non-null    float64  
 2   x        7734 non-null    float64  
 3   Value   7734 non-null    float32  
dtypes: float32(1), float64(2), int64(1)
memory usage: 271.9 KB
```

The table we have built has no geometries associated with it, only rows representing pixels. It takes a bit more effort, but it is possible to convert it, or a subset of it, into a fully-fledged geographic table, where each pixel includes the grid geometry it represents. For this task, we develop a function that takes a row from our table and the resolution of the surface, and returns its geometry:

```
def row2cell(row, res_xy):
    res_x, res_y = res_xy # Extract resolution for each dimension
    # XY Coordinates are centered on the pixel
    minX = row["x"] - (res_x / 2)
    maxX = row["x"] + (res_x / 2)
    minY = row["y"] - (res_y / 2)
    maxY = row["y"] + (res_y / 2)
    poly = geometry.box(
        minX, minY, maxX, maxY
    ) # Build squared polygon
    return poly
```

For example:

```
row2cell(t_surface.loc[0, :], surface.attrs["res"])
```

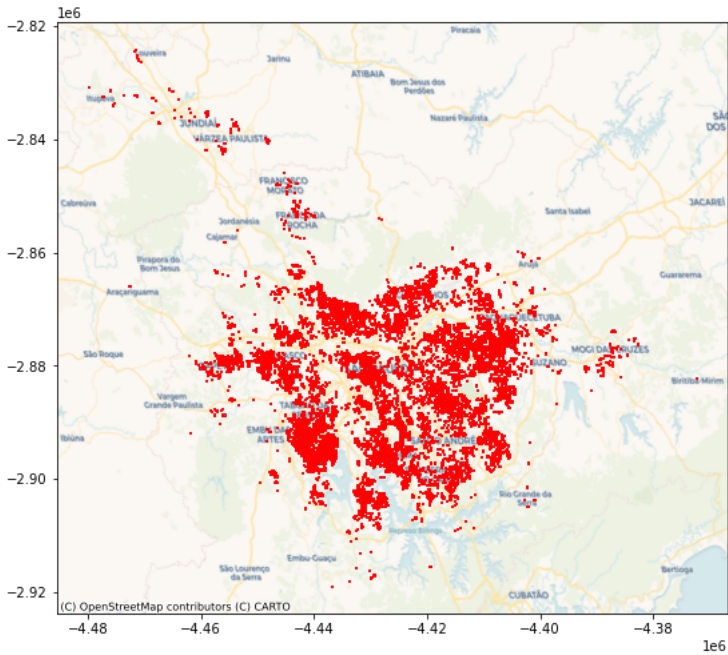


One of the benefits of this approach is we do not require entirely filled surfaces and can only record pixels where we have data. For the example above or cells with more than 1,000 people, we could create the associated geo-table as follows:

```
max_polys = (
    t_surface.query(
        "Value > 1000"
    ) # Keep only cells with more than 1k people
    .apply( # Build polygons for selected cells
        row2cell, res_xy=surface.attrs["res"], axis=1
    )
    .pipe( # Pipe result from apply to convert into a GeoSeries
        geopandas.GeoSeries, crs=surface.attrs["crs"]
    )
)
```

And generate a map with the same tooling that we use for any standard geo-table:

```
# Plot polygons
ax = max_polys.plot(edgecolor="red", figsize=(9, 9))
# Add basemap
cx.add_basemap(
    ax, crs=surface.attrs["crs"], source=cx.providers.CartoDB.Voyager
);
```



Finally, once we have operated on the data as a table, we may want to return to a surface-like data structure. This involves taking the same journey in the opposite direction as how we started. The sister method of `to_series` in `xarray` is `from_series`:

```
new_da = xarray.DataArray.from_series(
    t_surface.set_index(["band", "y", "x"])[ "Value" ]
)
new_da
```

xarray.DataArray 'Value' (band: 1, y: 416, x: 468)

array([[[-200., -200., -200., ..., -200., -200., -200.],  
 [-200., -200., -200., ..., -200., -200., -200.],  
 [-200., -200., -200., ..., -200., -200., -200.],  
 ...,  
 [-200., -200., -200., ..., -200., -200., -200.],  
 [-200., -200., -200., ..., -200., -200., -200.],  
 [-200., -200., -200., ..., -200., -200., -200.]]],  
 dtype=float32)

▼ Coordinates:

band	(band)	int64	1	
y	(y)	float64	-2.926e+06 ... -2.822e+06	
x	(x)	float64	-4.482e+06 ... -4.365e+06	

► Attributes: (0)

Pixels to polygons

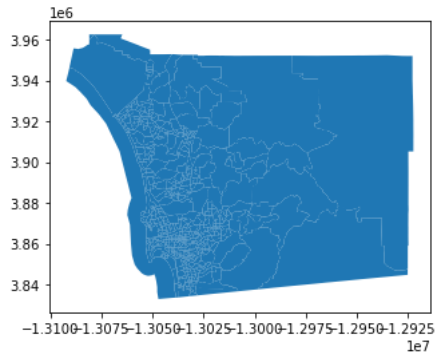
A second use case involves moving surfaces directly into geographic tables by aggregating pixels into pre-specified geometries. For this illustration, we will use the [DEM](#) surface containing elevation for the San Diego (US) region, and the set of [Census tracts](#). For an example, we will investigate the average altitude of each neighborhood.

Let’s start by reading the data. First, the elevation model:

```
dem = xarray.open_rasterio("../data/nasadem/nasadem_sd.tif").sel(
    band=1
)
dem.where(dem > 0).plot.imshow();
```

And the neighborhood areas (tracts) from the Census:

```
sd_tracts = geopandas.read_file(
    "../data/sandiego/sandiego_tracts.gpkg"
)
sd_tracts.plot();
```



There are several approaches to compute the average altitude of each neighborhood. We will use `rioarray` to clip parts of the surface *within* a given set of geometries. By this, we mean that we will cut out the part of the raster that falls within each geometry, and then we can summarize the values in that sub-raster. This is sometimes called computing a “zonal statistic” from a raster, where the “zone” is the geometry.

Since this is somewhat complicated, we will start with a single polygon. For the illustration, we will use the largest one, located on the eastern side of San Diego. We can find the ID of the polygon with:

```
largest_tract_id = sd_tracts.query(
    f"area_sqm == {sd_tracts['area_sqm'].max()}"
).index[0]

largest_tract_id
```

627

And then pull out the polygon itself for the illustration:

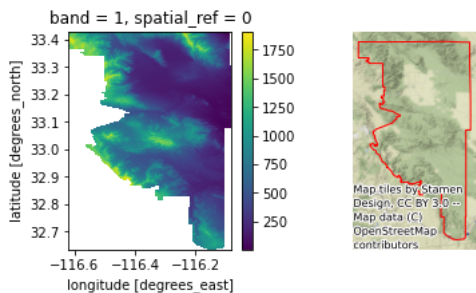
```
largest_tract = sd_tracts.loc[largest_tract_id, "geometry"]
```

Clipping the section of the surface that is within the polygon in the DEM can be achieved with the `rioarray` extension to clip surfaces based on geometries:

```
# Clip elevation for largest tract
dem_clip = dem.rio.clip(
    [largest_tract.__geo_interface__], crs=sd_tracts.crs
)

# Set up figure to display against polygon shape
f, axs = plt.subplots(1, 2, figsize=(6, 3))
# Display elevation of largest tract
dem_clip.where(dem_clip > 0).plot(ax=axs[0], add_colorbar=True)


# Display largest tract polygon
sd_tracts.loc[[largest_tract_id]].plot(
    ax=axs[1], edgecolor="red", facecolor="none"
)
axs[1].set_axis_off()
# Add basemap
cx.add_basemap(
    axs[1], crs=sd_tracts.crs, source=cx.providers.Stamen.Terrain
);
```



Once we have elevation measurements for all the pixels within the tract, the average one can be calculated with `mean()`:

```
dem_clip.where(dem_clip > 0).mean()
```

xarray.DataArray

 array(585.11375946)

▼ Coordinates:

band

() int64 1

spatial\_ref

() int64 0

► Attributes: (0)

Now, to scale this to the entire geo-table, there are several approaches. Each has its benefits and disadvantages. We opt for applying the method above to each row of the table. We define an auxiliary function that takes a row containing one of our tracts and returns its elevation:

```
def get_mean_elevation(row, dem):
    # Extract geometry object
    geom = row["geometry"].__geo_interface__
    # Clip the surface to extract pixels within `geom`
    section = dem.rio.clip([geom], crs=sd_tracts.crs)
    # Calculate mean elevation
    elevation = float(section.where(section > 0).mean())
    return elevation
```

Applied to the same tract, it returns the same average elevation:

```
get_mean_elevation(sd_tracts.loc[largest_tract_id, :], dem)

585.1137594576915
```

This method can then be run on each polygon in our series using the `apply()` method:

```
elevations = sd_tracts.head().apply(
    get_mean_elevation, dem=dem, axis=1
)
elevations

0      7.144268
1     35.648492
2     53.711389
3     91.358777
4    187.311972
dtype: float64
```

This simple approach illustrates the main idea well: find the cells that pertain to a given geometry and summarize their values in some manner. This can be done with any kind of geometry. Further, this simple method plays well with `xarray` surface structures and is scalable in that it is not too involved to run in parallel and distributed form using libraries like `dask`. Further, it can be extended using arbitrary Python functions, so it is simple to extend.

However, this approach can be quite slow in big data. A more efficient alternative for our example uses the `rasterstats` library. This is a purpose-built library to construct so-called “zonal statistics” from surfaces. Here, the “zones” are the polygons and the “surface” is our `DataArray`. Generally, this library will be faster than the simpler approach used above, but may be more difficult to extend or adapt:

```
from rasterstats import zonal_stats

elevations2 = zonal_stats(
    sd_tracts.to_crs(dem.rio.crs), # Geotable with zones
    "../data/nasadem/nasadem_sd.tif", # Path to surface file
)
elevations2 = pandas.DataFrame(elevations2)

elevations2.head()
```

	min	max	mean	count
0	-12.0	18.0	3.538397	3594
1	-2.0	94.0	35.616395	5709
2	-5.0	121.0	48.742630	10922
3	31.0	149.0	91.358777	4415
4	-32.0	965.0	184.284941	701973

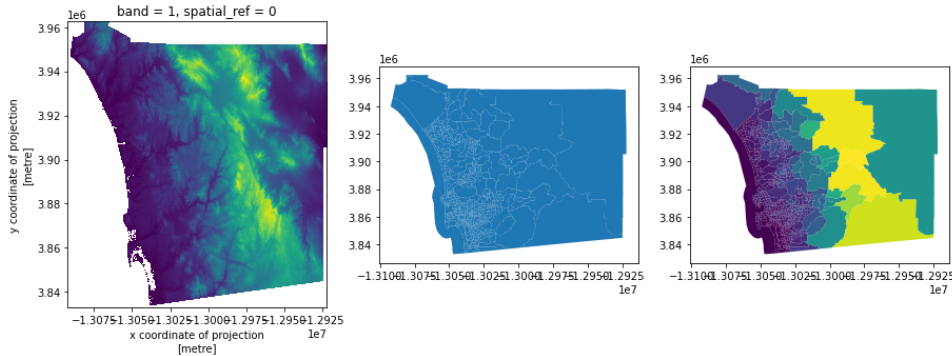
To visualize these results, we can make an elevation map:

```
# Set up figure
f, axs = plt.subplots(1, 3, figsize=(15, 5))

# Plot elevation surface
dem.where( # Keep only pixels above sea level
    dem
    > 0
    # Reproject to CRS of tracts
).rio.reproject(
    sd_tracts.crs
    # Render surface
).plot.imshow(
    ax=axs[0], add_colorbar=False
)

# Plot tract geography
sd_tracts.plot(ax=axs[1])

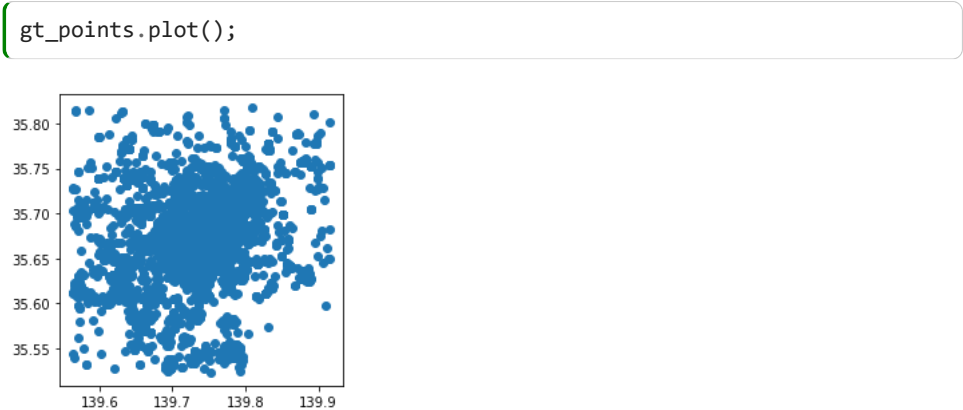
# Plot elevation on tract geography
sd_tracts.assign( # Append elevation values to tracts
    elevation=elevations2["mean"]
).plot( # Plot elevation choropleth
    "elevation", ax=axs[2]
);
```



Tables as surfaces

O caso de converter tabelas em superfícies talvez seja menos controverso do que transformar superfícies em tabelas. Essa é uma abordagem que podemos adotar nos casos em que estamos interessados na distribuição *geral* dos objetos (geralmente pontos) e temos tantos que não é apenas tecnicamente mais eficiente representá-los como uma superfície, mas conceitualmente também é mais fácil pense nos pontos como medições desiguais de um campo contínuo. Para ilustrar essa abordagem, usaremos o conjunto de dados de [fotografias de Tóquio](#) que carregamos acima em `gt_points`.

De uma perspectiva puramente técnica, para conjuntos de dados com muitos pontos, representar cada ponto dos dados em uma tela pode ficar seriamente superlotado:



Nesta figura, é difícil dizer algo sobre a densidade de pontos no centro da imagem devido ao *overplotting* : embora os pontos *teoricamente* não tenham largura, eles *devem* ter alguma dimensão para que possamos vê-los! Portanto, *marcadores de pontos* muitas vezes plotam um em cima do outro, obscurecendo o verdadeiro padrão e densidade em áreas densas. A conversão do conjunto de dados de uma geotabela em uma superfície envolve a criação de uma grade e a contagem de quantos pontos estão dentro de cada célula. Em certo sentido, esta é a operação inversa ao que vimos ao calcular as estatísticas zonais na seção anterior: em vez de agregar células em objetos, agregamos objetos em células. Ambas as operações, no entanto, envolvem agregações que reduzem a quantidade de informações presentes para tornar os (novos) dados mais gerenciáveis.

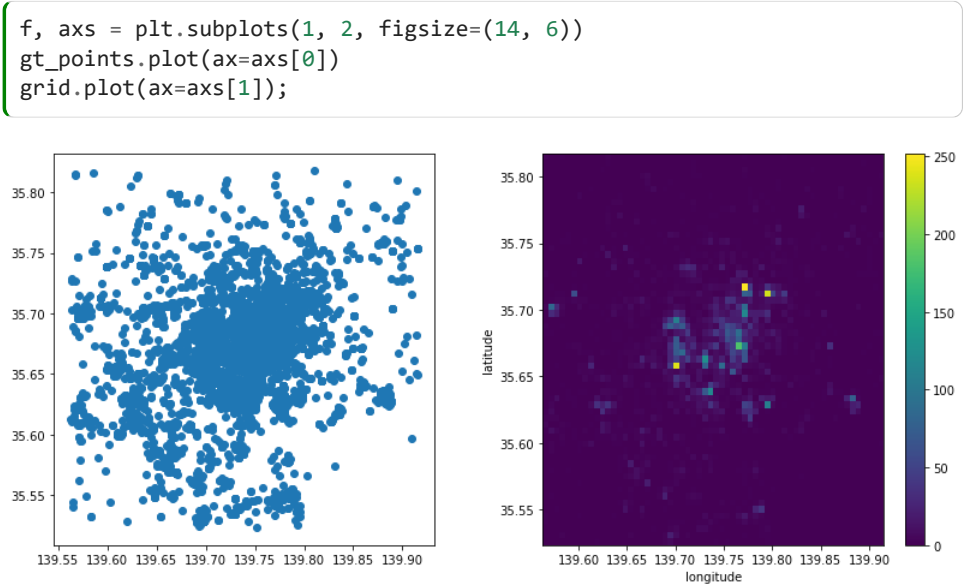
Em Python, podemos contar com a `datashader` biblioteca, que faz toda a computação de forma muito eficiente. Este processo envolve duas etapas principais. Primeiro, configuramos a grade (ou tela, `cvs`) na qual queremos agregar pontos:

```
cvs = datashader.Canvas(plot_width=60, plot_height=60)
```

Em seguida, “transferimos” os pontos para a grade:

```
grid = cvs.points(gt_points, x="longitude", y="latitude")
```

O resultado `grid` é um `DataArray` objeto padrão que podemos manipular como vimos antes. Quando plotados abaixo, a quantidade de detalhes que os dados reamostrados permitem é muito maior do que quando os pontos foram visualizados sozinhos. Isso é mostrado na Figura 14.



Redes como gráficos e tabelas

No capítulo anterior, vimos redes como estruturas de dados que armazenam *conexões* entre objetos. Também discutimos como essa definição ampla inclui muitas interpretações que se concentram em diferentes aspectos das redes. Embora a análise espacial possa usar gráficos para registrar a topologia de uma tabela de objetos, como polígonos; aplicações de transporte podem tratar a representação em rede do traçado da rua como um conjunto de objetos em si, neste caso linhas. Nesta seção final, mostramos como se pode alternar entre uma representação e outra, para aproveitar diferentes aspectos.



Começamos com o **graph** objeto da [seção anterior](#) . Lembre-se de que isso captura o layout das ruas ao redor do parque Yoyogi em Tóquio. Vimos como, armazenados sob essa estrutura de dados, é fácil consultar qual nó está conectado a qual e quais estão no final de uma determinada aresta.

No entanto, em alguns casos, podemos querer converter o grafo em uma estrutura que nos permita operar em cada componente da rede de forma independente. Por exemplo, podemos querer mapear ruas, calcular comprimentos de segmentos ou desenhar buffers em torno de cada interseção. Estas são todas as operações que não requerem informações topológicas, que são padrão para geotabelas e que são irrelevantes para a estrutura do grafo. Nesse contexto, faz sentido converter nosso **graph** em duas tabelas geográficas, uma para interseções (os nós do gráfico) e outra para segmentos de rua (as arestas do gráfico). Em **osmnx**, podemos fazer isso com o conversor embutido:

```
gt_intersections, gt_lines = osmnx.graph_to_gdfs(graph)
```

Agora, cada uma das tabelas geográficas resultantes é uma coleção de objetos geográficos:

```
gt_intersections.head()
```

	y	x	street_count	autoestrada	geometria
osmid					
	PONTO				
886196069	35.670087	139.694333	3	NaN	(139.69433 35.67009)
	PONTO				
886196073	35.669725	139.699508	3	NaN	(139.69951 35.66972)
	PONTO				
886196100	35.669442	139.699708	3	NaN	(139.69971 35.66944)
	PONTO				
886196106	35.670422	139.698564	4	NaN	(139.69856 35.67042)
	PONTO				
886196117	35.671256	139.697470	3	NaN	(139.69747 35.67126)

```
gt_lines.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
MultiIndex: 287 entries, (886196069, 1520546857, 0) to (7684088896,
3010293702, 0)
Data columns (total 8 columns):
#   Column      Non-Null Count  Dtype
---  -
0   osmid       287 non-null    object
1   highway     287 non-null    object
2   oneway      287 non-null    bool
3   length      287 non-null    float64
4   geometry    287 non-null    geometry
5   bridge      8 non-null      object
6   name        9 non-null      object
7   access      2 non-null      object
dtypes: bool(1), float64(1), geometry(1), object(5)
memory usage: 26.9+ KB
```

Se estivéssemos na situação oposta, onde tivéssemos um conjunto de segmentos de rua e suas interseções em forma de geotabela, poderíamos gerar a representação gráfica com o **graph\_from\_gdfs** método irmão:

```
new_graph = osmnx.graph_from_gdfs(gt_intersections, gt_lines)
```

O objeto resultante se comportará da mesma forma que nosso original **graph**.

## Conclusão

Concluindo, este capítulo fornece uma visão geral dos mapeamentos entre modelos de dados, apresentados no Capítulo 2, e estruturas de dados comuns em Python. Além das estruturas de dados discutidas aqui, o ecossistema Python é vasto, profundo e em constante mudança. Parte disso é a facilidade com que você pode criar suas próprias representações para expressar diferentes aspectos de um problema em questão. No entanto, concentrando-se em nossas representações compartilhadas e nas interfaces entre essas representações, você geralmente pode realizar qualquer análise necessária. Ao criar representações exclusivas e personalizadas, sua análise pode ser mais eficiente, mas você também pode isolá-la inadvertidamente de outros desenvolvedores e tornar ferramentas úteis inoperáveis. Portanto, uma sólida compreensão das estruturas básicas de dados (o **GeoDataFrame**, **DataArray** e **Graph**) será suficiente para dar suporte a praticamente qualquer análise que você precise realizar.

## Perguntas

1. Uma maneira de converter **Multi**-geometrias de tipo em muitas geometrias individuais é usar o **explode()** método de um `GeoDataFrame`. Usando o **explode()** método, você pode descobrir quantas ilhas existem na Indonésia?
2. Usando **osmnx**, você consegue extrair o gráfico de ruas da sua cidade natal?
3. Como você viu com o **osmnx.graph\_to\_gdfs()** método, é possível converter um grafo nos nós e arestas constituintes. Os gráficos têm muitos outros tipos de representações não geográficas. Muitos deles são fornecidos em **networkx** métodos que começam com **to\_**. Quantas representações de gráficos são suportadas atualmente?
4. Usando **networkx.to\_edgelist()**, quais informações “extras” **osmnx** incluem ao construir o dataframe para arestas?
5. Em vez de calcular a elevação média de cada bairro em San Diego, você pode responder às seguintes perguntas?

- Qual bairro (ou bairros) tem *a maior elevação média* ?
- Qual bairro (ou bairros) tem *o ponto único mais alto* ?
- Você consegue encontrar o bairro (ou bairros) com *a maior mudança de elevação* ?

- 
- [1].

Geralmente usaremos dois colchetes curvos (como **method\_name()**) para denotar uma *função* e os omitiremos (como **package**) ao nos referirmos a um objeto ou pacote.]

[2].

Embora, se tudo o que você deseja fazer é esse tipo de consulta, **xarray** também está bem equipado para esse tipo de tarefa.

---
- Por Sergio J. Rey, Dani Arribas-Bel, Levi J. Wolf  
© Copyright 2020. See More
- 
- Este trabalho está licenciado sob uma [Licença Creative Commons Atribuição-NãoComercial-SemDerivações 4.0 Internacional](#).
- https://geographicdata.science/book/notebooks/03\_spatial\_data.html

15/15