THE PYTHON SERIES

GEOCOMPUTATION WITH PYTHON

MICHAEL DORMAN, ANITA GRASER,
JAKUB NOWOSAD, AND ROBIN LOVELACE

A **Chapman & Hall** Book

CRC Press
Taylor & Francis Group

Geocomputation with Python

Geocomputation with Python is a comprehensive resource for working with geographic data with the most popular programming language in the world. The book gives an overview of Python's capabilities for spatial data analysis, as well as dozens of worked-through examples covering the entire range of standard GIS operations. A unique selling point of the book is its cohesive and joined-up coverage of both vector and raster geographic data models and consistent learning curve. This book is an excellent starting point for those new to working with geographic data with Python, making it ideal for students and practitioners beginning their journey with Python.

Key features:

- Showcases the integration of vector and raster datasets operations.
- Provides explanation of each line of code in the book to minimize surprises.
- Includes example datasets and meaningful operations to illustrate the applied nature of geographic research.

Another unique feature is that this book is part of a wider community. *Geocomputation with Python* is a sister project of *Geocomputation with R* (Lovelace, Nowosad, and Muenchow 2019), a book on geographic data analysis, visualization, and modeling using the R programming language that has numerous contributors and an active community.

The book teaches how to import, process, examine, transform, compute, and export spatial vector and raster datasets with Python, the most widely used language for data science and many other domains. Reading the book and running the reproducible code chunks within will make you a proficient user of key packages in the ecosystem, including shapely, geopandas, and rasterio. The book also demonstrates how to make use of dozens of additional packages for a wide range of tasks, from interactive map making to terrain modeling. Geocomputation with Python provides a firm foundation for more advanced topics, including spatial statistics, machine learning involving spatial data, and spatial network analysis, and a gateway into the vibrant and supportive community developing geographic tools in Python and beyond.

Chapman & Hall/CRC
The Python Series

About the Series

Python has been ranked as the most popular programming language, and it is widely used in education and industry. This book series will offer a wide range of books on Python for students and professionals. Titles in the series will help users learn the language at an introductory and advanced level, and explore its many applications in data science, AI, and machine learning. Series titles can also be supplemented with Jupyter notebooks.

Statistics and Data Visualisation with Python
Jesús Rogel-Salazar

Introduction to Python for Humanists
William J.B. Mattingly

Python for Scientific Computation and Artificial Intelligence
Stephen Lynch

Learning Professional Python Volume 1: The Basics
Usharani Bhimavarapu and Jude D. Hemanth

Learning Professional Python Volume 2: Advanced
Usharani Bhimavarapu and Jude D. Hemanth

Learning Advanced Python from Open Source Projects
Rongpeng Li

Foundations of Data Science with Python
John Mark Shea

Data Mining with Python: Theory, Applications, and Case Studies
Di Wu

A Simple Introduction to Python
Stephen Lynch

Introduction to Python: with Applications in Optimization, Image and Video Processing, and Machine Learning
David Baez-Lopez and David Alfredo Báez Villegas

Tidy Finance with Python
Christoph Frey, Christoph Scheuch, Stefan Voigt and Patrick Weiss

Introduction to Quantitative Social Science with Python
Weiqi Zhang and Dmitry Zinoviev

Python Programming for Mathematics
Julien Guillod

Geocomputation with Python
Michael Dorman, Anita Graser, Jakub Nowosad and Robin Lovelace

For more information about this series please visit: https://www.routledge.com/Chapman--HallCRC-The-Python-Series/book-series/PYTH

Geocomputation with Python

Michael Dorman, Anita Graser, Jakub Nowosad,
and Robin Lovelace

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Designed cover image: © Michael Dorman

First edition published 2025
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

ISBN: 978-1-032-45891-5 (hbk)
ISBN: 978-1-032-46065-9 (pbk)
ISBN: 978-1-003-37991-1 (ebk)

DOI: 10.1201/9781003379911

Typeset in Latin Modern font
by KnowledgeWorks Global Ltd.

Publisher's note: This book has been prepared from camera-ready copy provided by the authors.

For Ariel

For Marko

Dla Zosi i Czesi

For Katy and Kit

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Table of contents

Preface

Geocomputation with Python (*geocompy*) is motivated by the need for an introductory resource for working with geographic data with the most popular programming language in the world. A unique selling point of the book is its cohesive and joined-up coverage of *both vector and raster* geographic data models and consistent learning curve. We aim to *minimize surprises*, with each section and chapter building on the previous. If you're just starting out with Python for working with geographic data, this book is an excellent place to start.

There are many resources on Python on 'GeoPython' but none that fill this need for an introductory resource that provides strong foundations for future work. We want to avoid reinventing the wheel and provide something that fills an 'ecological niche' in the wider free and open-source software for geospatial (FOSS4G) ecosystem. Key features include:

1. Doing basic operations well
2. Integration of vector and raster datasets operations
3. Clear explanation of each line of code in the book to minimize surprises
4. Provision of lucid example datasets and meaningful operations to illustrate the applied nature of geographic research

This book complements and adds value to other projects in the ecosystem, as highlighted in the following comparison between *Geocomputation with Python* and related GeoPython books:

- *Learning Geospatial Analysis with Python*[1] and *Geoprocessing with Python*[2] are books in this space that focus on processing spatial data using low-level Python interfaces for GDAL, such as the **gdal**, **gdalnumeric**, and **ogr** packages from **osgeo**. This approach requires writing more lines of code. We believe our approach is more 'Pythonic' and future-proof, in light of development of packages such as **geopandas** and **rasterio**.

[1] https://www.packtpub.com/product/learning-geospatial-analysis-with-python/9781783281138

[2] https://www.manning.com/books/geoprocessing-with-python

- *Introduction to Python for Geographic Data Analysis*[3] (in progress) seeks to provide a general introduction to 'GIS in Python', with parts focusing on Python essentials, using Python with GIS, and case studies. Compared with this book, which is also open source, and is hosted at pythongis.org, *Geocomputation with Python* has a narrower scope (not covering spatial network analysis, for example) and more coverage of raster data processing and raster-vector interoperability.
- *Geographic Data Science with Python*[4] is an ambitious project with chapters dedicated to advanced topics, with Chapter 4 on Spatial Weights getting into complex topics relatively early, for example.
- *Python for Geospatial Data Analysis*[5] introduces a wide range of approaches to working with geospatial data using Python, including automation of proprietary and open-source GIS software, as well as standalone open-source Python packages (which is what we focus on and explain comprehensively in our book). Geocompy is shorter, simpler and more introductory, and covers raster and vector data with equal importance.

Another unique feature of the book is that it is part of a wider community. *Geocomputation with Python* is a sister project of *Geocomputation with R*[6](Lovelace, Nowosad, and Muenchow 2019), a book on geographic data analysis, visualization, and modeling using the R programming language that has 60+ contributors and an active community, not least in the associated Discord group[7]. Links with the vibrant 'R-spatial' community, and other communities such as GeoRust and JuliaGeo, lead to many opportunities for mutual benefit across open-source ecosystems.

Prerequisites

We assume that the reader is:

- familiar with the Python language,
- is capable of running Python code and install Python packages, and
- is familiar with the `numpy` and `pandas` packages for working with data in Python.

From that starting point on, the book introduces the topic of working with *spatial data* in Python, through dedicated third-party packages—most importantly `geopandas` and `rasterio`.

[3] https://pythongis.org
[4] https://geographicdata.science/book/intro.html
[5] https://www.oreilly.com/library/view/python-for-geospatial/9781098104788/
[6] https://r.geocompx.org/
[7] https://discord.gg/PMztXYgNxp

We also assume familiarity with theoretical concepts of geographic data and GIS, such as coordinate systems, projections, spatial layer file formats, etc., which is necessary for understanding the reasoning of the examples.

Code and sample data

To run the code examples, you can download[8] the ZIP file of the GitHub repository. In the ZIP file, the `ipynb` directory contains the source files of the chapters in Jupyter Notebook format, the `data` directory contains the sample data files, and the `output` directory contains the files created in code examples (some of which are also used as inputs in other code sections). Place them together as follows to run the code:

```
├── data
│   ├── aut.tif
│   ├── ch.tif
│   ├── coffee_data.csv
│   ├── cycle_hire.gpkg
│   ├── cycle_hire_osm.gpkg
│   ├── cycle_hire_xy.csv
│   ├── dem.tif
│   ├── landsat.tif
│   ├── nlcd.tif
│   ├── nz_elev.tif
│   ├── nz.gpkg
│   ├── nz_height.gpkg
│   ├── seine.gpkg
│   ├── srtm.tif
│   ├── us_states.gpkg
│   ├── world.gpkg
│   ├── world_wkt.csv
│   ├── zion.gpkg
│   └── zion_points.gpkg
├── output
│   ├── cycle_hire_xy.csv
│   ├── dem_agg5.tif
│   ├── dem_contour.gpkg
│   ├── dem_resample_maximum.tif
│   ├── dem_resample_nearest.tif
│   ├── elev.tif
│   ├── grain.tif
```

[8] https://github.com/geocompx/geocompy/zipball/master

```
│   ├── map.html
│   ├── ne_10m_airports.cpg
│   ├── ne_10m_airports.dbf
│   ├── ne_10m_airports.prj
│   ├── ne_10m_airports.README.html
│   ├── ne_10m_airports.shp
│   ├── ne_10m_airports.shx
│   ├── ne_10m_airports.VERSION.txt
│   ├── ne_10m_airports.zip
│   ├── nlcd_4326_2.tif
│   ├── nlcd_4326.tif
│   ├── nlcd_modified_crs.tif
│   ├── plot_geopandas.jpg
│   ├── plot_rasterio2.svg
│   ├── plot_rasterio.jpg
│   ├── r3.tif
│   ├── r_nodata_float.tif
│   ├── r_nodata_int.tif
│   ├── r.tif
│   ├── srtm_32612_aspect.tif
│   ├── srtm_32612_slope.tif
│   ├── srtm_32612.tif
│   ├── srtm_masked_cropped.tif
│   ├── srtm_masked.tif
│   ├── w_many_features.gpkg
│   ├── w_many_layers.gpkg
│   └── world.gpkg
├── 01-spatial-data.ipynb
├── 02-attribute-operations.ipynb
├── 03-spatial-operations.ipynb
├── 04-geometry-operations.ipynb
├── 05-raster-vector.ipynb
├── 06-reproj.ipynb
├── 07-read-write.ipynb
└── 08-mapping.ipynb
```

Software

Python version used when rendering the book:

```
3.12.6 (main, Sep  9 2024, 00:00:00)
        [GCC 14.2.1 20240801 (Red Hat 14.2.1-1)]
```

Versions of the main packages used in the book:

```
numpy==2.0.1
pandas==2.2.2
shapely==2.0.5
geopandas==1.0.1
rasterio==1.3.10
matplotlib==3.9.0
rasterstats==0.19.0
```

Acknowledgments

We acknowledge Robin Lovelace, Jakub Nowosad, and Jannes Muenchow—authors of *Geocomputation with R* (Robin and Jakub also author the present book), a book on the same topic for a different programming language (R). The structure, topics, and most of the theoretical discussions were adapted from that earlier publication.

We thank the authors of the Python language, and the authors of the **numpy**, **pandas**, **shapely**, **geopandas**, and **rasterio** packages which are used extensively in the book, for building these wonderful tools.

We acknowledge GitHub users Will Deakin, Sean Gillies, Josh Cole, Jt Miclat, and Zehui Yin (at the time of writing; full list on GitHub[9]) for their contributions during the open-source development of the book.

[9] https://github.com/geocompx/geocompy/graphs/contributors

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Authors

Michael Dorman, Ph.D. is a programmer and lecturer at the Department of Environmental, Geoinformatics and Urban Planning Sciences, Ben-Gurion University of the Negev. He is working with researchers and students to develop computational workflows for spatial analysis, mostly through programming in Python, R, and JavaScript, as well as teaching those subjects.

Anita Graser, Ph.D. is a Senior Scientist at the Austrian Institute of Technology (AIT), QGIS PSC member and lead developer of MovingPandas. Anita has published several books about QGIS, including "Learning QGIS" and "QGIS Map Design", teaches Python for QGIS, and writes a popular spatial data science blog.

Jakub Nowosad, Ph.D. is an Associate Professor at Adam Mickiewicz University in Poznań and a visiting scientist at the University of Münster. Specializing in spatial pattern analysis in environmental studies, he combines research with a dedication to education and open science principles. Dr. Nowosad is committed to developing scientific software and fostering accessible knowledge through teaching and open-source contributions.

Robin Lovelace, Ph.D. is a Professor of Transport Data Science at the University of Leeds. He is the developer of high impact applications for more data-driven transport planning and policy. He has a decade's experience researching and teaching data science with geographic data and has developed numerous tools to support more data-driven policies, including the award-winning Propensity to Cycle Tool which has transformed the practice of strategic active travel network planning in the UK.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

1

Geographic data in Python

1.1 Introduction

This chapter outlines two fundamental geographic data models (vector and raster) and introduces Python packages for working with them. Before demonstrating their implementation in Python, we will introduce the theory behind each data model and the disciplines in which they predominate.

The vector data model (Section 1.2) represents geographic entities with points, lines, and polygons. These have discrete, well-defined borders, meaning that vector datasets usually have a high level of precision (but not necessarily accuracy). The raster data model (Section 1.3), on the other hand, divides the surface up into cells of constant size. Raster datasets are the basis of background images used in online maps and have been a vital source of geographic data since the origins of aerial photography and satellite-based remote sensing devices. Rasters aggregate spatially specific features to a given resolution, meaning that they are consistent over space and scalable, with many worldwide raster datasets available.

Which to use? The answer likely depends on your domain of application, and the datasets you have access to:

- Vector datasets and methods dominate the social sciences because human settlements and processes (e.g., transport infrastructure) tend to have discrete borders
- Raster datasets and methods dominate many environmental sciences because of the reliance on remote sensing data

Python has strong support for both data models. We will focus on **shapely** and **geopandas** for working with geographic vector data, and **rasterio** for working with rasters.

shapely is a 'low-level' package for working with individual vector geometry objects. **geopandas** is a 'high-level' package for working with geometry columns (`GeoSeries` objects), which internally contain **shapely** geometries, and with vector layers (`GeoDataFrame` objects). The **geopandas** ecosystem provides a comprehensive approach for working with vector layers in Python, with many packages building on it.

DOI: 10.1201/9781003379911-1

There are several partially overlapping packages for working with raster data, each with its own advantages and disadvantages. In this book, we focus on the most prominent one: **rasterio**, which represents 'simple' raster datasets with a combination of a **numpy** array, and a metadata object (`dict`) providing geographic metadata such as the coordinate system. **xarray** is a notable alternative to **rasterio** not covered in this book which uses native `xarray.Dataset` and `xarray.DataArray` classes to effectively represent complex raster datasets such as NetCDF files with multiple bands and metadata.

There is much overlap in some fields, and raster and vector datasets can be used together: ecologists and demographers, for example, commonly use both vector and raster data. Furthermore, it is possible to convert between the two forms (see Chapter 5). Whether your work involves use of vector or raster datasets, it is worth understanding the underlying data models before using them, as discussed in subsequent chapters.

1.2 Vector data

The geographic vector data model is based on points located within a coordinate reference system (CRS). Points can represent self-standing features (e.g., the location of a bus stop), or they can be linked together to form more complex geometries such as lines and polygons. Most point geometries contain only two dimensions (three-dimensional CRSs may contain an additional z value, typically representing height above sea level).

In this system, London, for example, can be represented by the coordinates `(-0.1,51.5)`. This means that its location is -0.1 degree east and 51.5 degree north of the origin. The origin, in this case, is at 0 degree longitude (a prime meridian located at Greenwich) and 0 degree latitude (the Equator) in a geographic ('lon/lat') CRS (Figure 1.1, left panel). The same point could also be approximated in a projected CRS with 'Easting/Northing' values of `(530000,180000)` in the British National Grid, meaning that London is located 530 km East and 180 km North of the origin of the CRS (Figure 1.1, right panel). The location of National Grid's origin, in the sea beyond South West Peninsular, ensures that most locations in the UK have positive Easting and Northing values.

There is more to CRSs, as described in Section 1.4 and Chapter 6 but, for the purposes of this section, it is sufficient to know that coordinates consist of two numbers representing the distance from an origin, usually in x and y dimensions.

geopandas (Bossche et al. 2023) provides classes for geographic vector data and a consistent command-line interface for reproducible geographic data

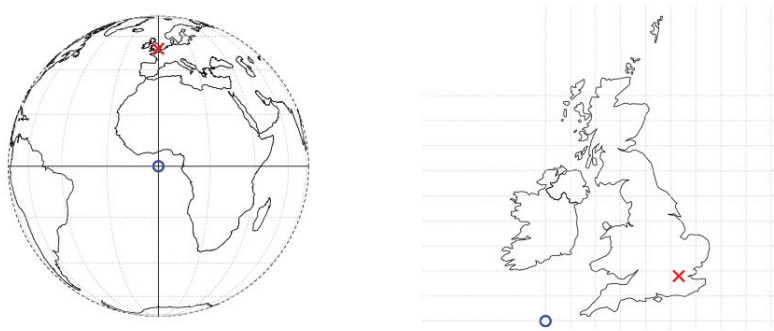


Figure 1.1: Illustration of vector (point) data in which location of London (the red X) is represented with reference to an origin (the blue circle). The left plot represents a geographic CRS with an origin at 0° longitude and latitude. The right plot represents a projected CRS with an origin located in the sea west of the South West Peninsula.

analysis in Python. It also provides an interface to three mature libraries for geocomputation, a strong foundation on which many geographic applications are built:

- GDAL, for reading, writing, and manipulating a wide range of geographic data formats, covered in Chapter 7
- PROJ, a powerful library for coordinate system transformations, which underlies the content covered in Chapter 6
- GEOS, a planar geometry engine for operations such as calculating buffers and centroids on data with a projected CRS, covered in Chapter 4

Tight integration with these geographic libraries makes reproducible geocomputation possible: an advantage of using a higher-level language such as Python to access these libraries is that you do not need to know the intricacies of the low-level components, enabling focus on the methods rather than the implementation.

1.2.1 Vector data classes

The main classes for working with geographic vector data in Python are hierarchical, meaning that the 'vector layer' class is composed of simpler 'geometry column' and individual 'geometry' components. This section introduces them in order, starting with the highest level class. For many applications, the vector layer class, a data frame with geometry columns, is all that's needed. However,

it's important to understand the structure of vector geographic objects and their components for some applications and for a deep understanding. The three main vector geographic data classes in Python are:

- `GeoDataFrame`, a class representing vector layers, with a geometry column (class `GeoSeries`) as one of the columns
- `GeoSeries`, a class that is used to represent the geometry column in `GeoDataFrame` objects
- `shapely` geometry objects, which represent individual geometries, such as a point or a polygon in `GeoSeries` objects

The first two classes (`GeoDataFrame` and `GeoSeries`) are defined in **geopandas**. The third class is defined in the **shapely** package, which deals with individual geometries, and is a main dependency of the **geopandas** package.

1.2.2 Vector layers

The most commonly used geographic vector data structure is the vector layer. There are several approaches for working with vector layers in Python, ranging from low-level packages (e.g., **osgeo**, **fiona**) to the relatively high-level **geopandas** package that is the focus of this section. Before writing and running code for creating and working with geographic vector objects, we need to import **geopandas** (by convention as `gpd` for more concise code) and **shapely**.

```
import pandas as pd
import shapely
import geopandas as gpd
```

We also limit the maximum number of printed rows to six, to save space, using the `'display.max_rows'` option of **pandas**.

```
pd.set_option('display.max_rows', 6)
```

Projects often start by importing an existing vector layer saved as a GeoPackage (`.gpkg`) file, an ESRI Shapefile (`.shp`), or other geographic file format. The function `gpd.read_file` imports a GeoPackage file named `world.gpkg` located in the `data` directory of Python's working directory into a `GeoDataFrame` named `gdf`.

```
gdf = gpd.read_file('data/world.gpkg')
```

The result is an object of type (class) `GeoDataFrame` with 177 rows (features) and 11 columns, as shown in the output of the following code:

```
type(gdf)
```

```
geopandas.geodataframe.GeoDataFrame
```

```
gdf.shape
```

```
(177, 11)
```

The `GeoDataFrame` class is an extension of the `DataFrame` class from the popular **pandas** package (McKinney 2010). This means we can treat non-spatial attributes from a vector layer as a table, and process them using the ordinary, i.e., non-spatial, established function methods. For example, standard data frame subsetting methods can be used. The code below creates a subset of the `gdf` dataset containing only the country name and the geometry.

```
gdf = gdf[['name_long', 'geometry']]
gdf
```

	name_long	geometry
0	Fiji	MULTIPOLYGON (((-180 -16.55522,...
1	Tanzania	MULTIPOLYGON (((33.90371 -0.95,...
2	Western Sahara	MULTIPOLYGON (((-8.66559 27.656...
...	...	...
174	Kosovo	MULTIPOLYGON (((20.59025 41.855...
175	Trinidad and Tobago	MULTIPOLYGON (((-61.68 10.76, -...
176	South Sudan	MULTIPOLYGON (((30.83385 3.5091...

The following expression creates a subdataset based on a condition, such as equality of the value in the `'name_long'` column to the string `'Egypt'`.

```
gdf[gdf['name_long'] == 'Egypt']
```

	name_long	geometry
163	Egypt	MULTIPOLYGON (((36.86623 22, 36...

Finally, to get a sense of the spatial component of the vector layer, it can be plotted using the `.plot` method (Figure 1.2).

```
gdf.plot();
```

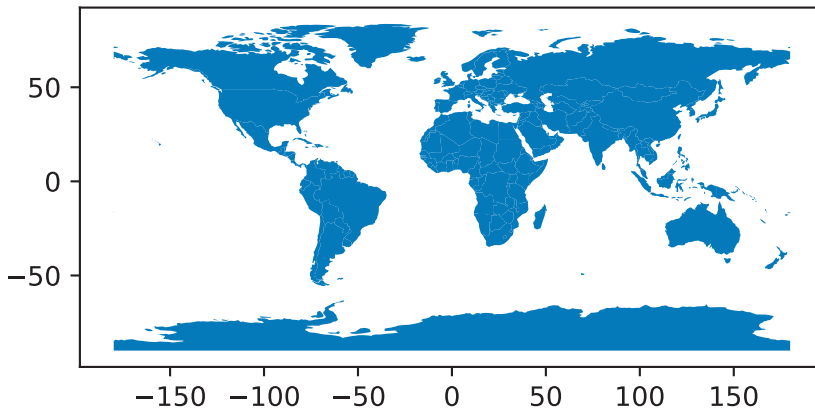


Figure 1.2: Basic plot of a `GeoDataFrame`

Interactive maps of `GeoDataFrame` objects can be created with the `.explore` method, as illustrated in Figure 1.3 which was created with the following command:

```
gdf.explore()
```

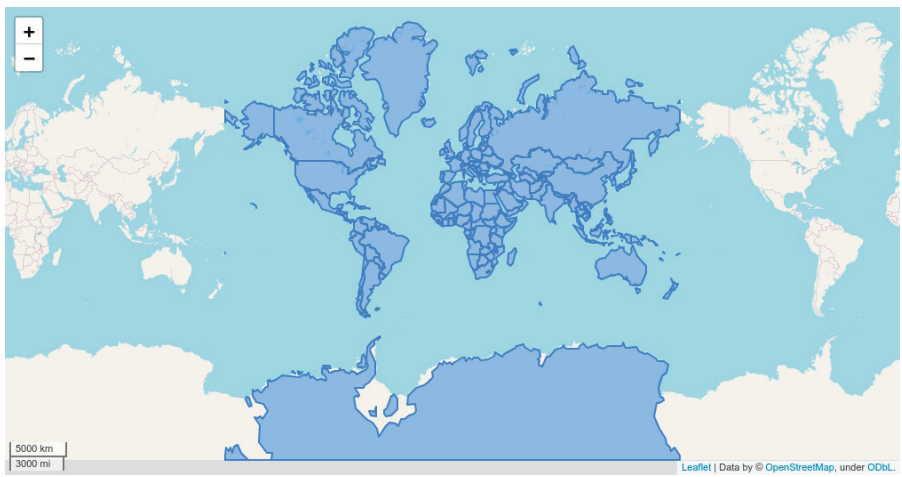


Figure 1.3: Basic interactive map with `.explore`

A subset of the data can be also plotted in a similar fashion.

```
gdf[gdf['name_long'] == 'Egypt'].explore()
```

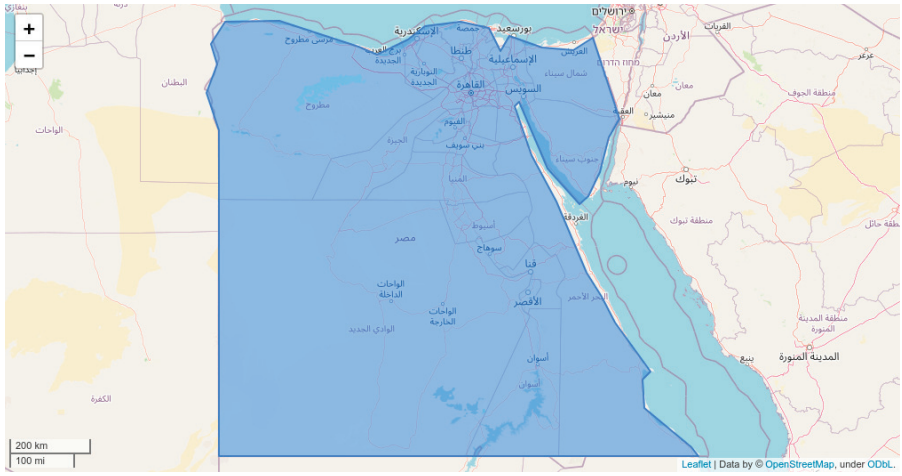


Figure 1.4: Interactive map of a `GeoDataFrame` subset

1.2.3 Geometry columns

The geometry column of class `GeoSeries` is an essential column in a `GeoDataFrame`. It contains the geometric part of the vector layer, and is the basis for all spatial operations. This column can be accessed by name, which typically (e.g., when reading from a file) is `'geometry'`, as in `gdf['geometry']`. However, the recommendation is to use the fixed `.geometry` property, which refers to the geometry column regardless whether its name is `'geometry'` or not. In the case of the `gdf` object, the geometry column contains `'MultiPolygon'`s associated with each country.

```
gdf.geometry
```

```
0      MULTIPOLYGON (((-180 -16.55522,...
1      MULTIPOLYGON (((33.90371 -0.95,...
2      MULTIPOLYGON (((-8.66559 27.656...
                    ...
174    MULTIPOLYGON (((20.59025 41.855...
175    MULTIPOLYGON (((-61.68 10.76, -...
176    MULTIPOLYGON (((30.83385 3.5091...
Name: geometry, Length: 177, dtype: geometry
```

The geometry column also contains the spatial reference information, if any (also accessible with the shortcut `gdf.crs`).

```
gdf.geometry.crs
```

```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
```

```
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

Many geometry operations, such as calculating the centroid, buffer, or bounding box of each feature, involve just the geometry. Applying this type of operation on a `GeoDataFrame` is therefore basically a shortcut to applying it on the `GeoSeries` object in the geometry column. For example, the two following commands return exactly the same result, a `GeoSeries` containing bounding box polygons (using the `.envelope` method).

```
gdf.envelope
```

```
0      POLYGON ((-180 -18.28799, 179.9...
1      POLYGON ((29.34 -11.72094, 40.3...
2      POLYGON ((-17.06342 20.99975, -...
                      ...
174    POLYGON ((20.0707 41.84711, 21....
175    POLYGON ((-61.95 10, -60.895 10...
176    POLYGON ((23.88698 3.50917, 35....
Length: 177, dtype: geometry
```

```
gdf.geometry.envelope
```

```
0      POLYGON ((-180 -18.28799, 179.9...
1      POLYGON ((29.34 -11.72094, 40.3...
2      POLYGON ((-17.06342 20.99975, -...
                      ...
174    POLYGON ((20.0707 41.84711, 21....
175    POLYGON ((-61.95 10, -60.895 10...
176    POLYGON ((23.88698 3.50917, 35....
Length: 177, dtype: geometry
```

Note that `.envelope`, and other similar operators in **geopandas** such as `.centroid` (Section 4.2.2), `.buffer` (Section 4.2.3), or `.convex_hull`, return only the geometry (i.e., a `GeoSeries`), not a `GeoDataFrame` with the original attribute data. In case we want the latter, we can create a copy of the `GeoDataFrame` and then 'overwrite' its geometry (or, we can overwrite the geometries directly in case we do not need the original ones, as in `gdf.geometry=gdf.envelope`).

```
gdf2 = gdf.copy()
gdf2.geometry = gdf.envelope
gdf2
```

	name_long	geometry
0	Fiji	POLYGON ((-180 -18.28799, 179.9...
1	Tanzania	POLYGON ((29.34 -11.72094, 40.3...
2	Western Sahara	POLYGON ((-17.06342 20.99975, -...
...	...	...
174	Kosovo	POLYGON ((20.0707 41.84711, 21....
175	Trinidad and Tobago	POLYGON ((-61.95 10, -60.895 10...
176	South Sudan	POLYGON ((23.88698 3.50917, 35....

Another useful property of the geometry column is the geometry type, as shown in the following code. Note that the types of geometries contained in a geometry column (and, thus, a vector layer) are not necessarily the same for every row. It is possible to have multiple geometry types in a single `GeoSeries`. Accordingly, the `.type` property returns a `Series` (with values of type `str`, i.e., strings), rather than a single value (the same can be done with the shortcut `gdf.geom_type`).

```
gdf.geometry.type
```

```
0      MultiPolygon
1      MultiPolygon
2      MultiPolygon
           ...
174    MultiPolygon
175    MultiPolygon
176    MultiPolygon
Length: 177, dtype: object
```

To summarize the occurrence of different geometry types in a geometry column, we can use the **pandas** `.value_counts` method. In this case, we see that the `gdf` layer contains only `'MultiPolygon'` geometries.

```
gdf.geometry.type.value_counts()
```

```
MultiPolygon    177
Name: count, dtype: int64
```

A `GeoDataFrame` can also have multiple `GeoSeries` columns, as demonstrated in the following code section.

```
gdf['bbox'] = gdf.envelope
gdf['polygon'] = gdf.geometry
gdf
```

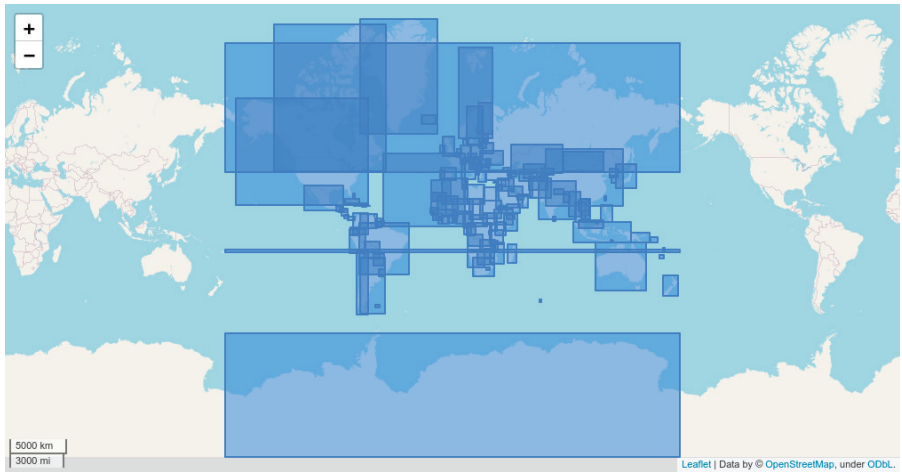


Figure 1.5: Switching to the `'bbox'` geometry column in the `world` layer, and plotting it

Only one geometry column at a time is 'active', in the sense that it is being accessed in operations involving the geometries (such as `.centroid`, `.crs`, etc.). To switch the active geometry column from one `GeoSeries` column to another, we use `.set_geometry`. Figure 1.5 and Figure 1.6 shows interactive maps of the `gdf` layer with the `'bbox'` and `'polygon'` geometry columns activated, respectively.

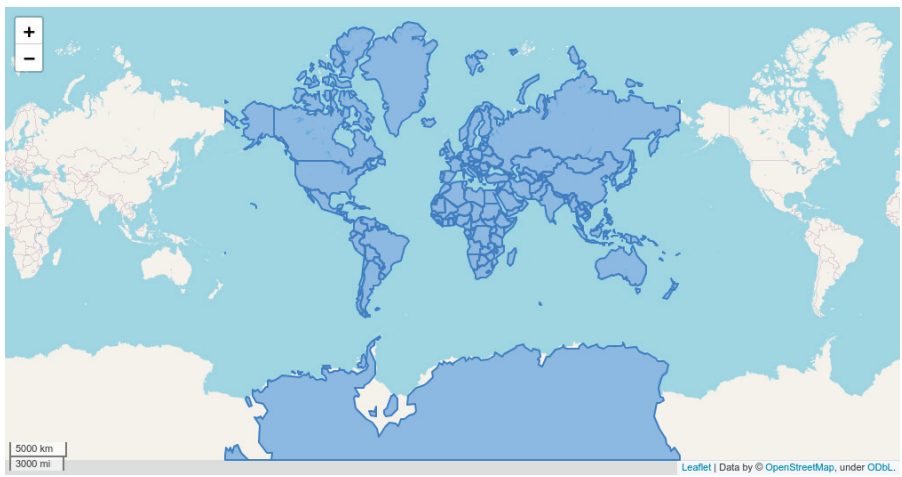


Figure 1.6: Switching to the `'polygons'` geometry column in the `world` layer, and plotting it

```
gdf = gdf.set_geometry('bbox')
gdf.explore()
```

```
gdf = gdf.set_geometry('polygon')
gdf.explore()
```

1.2.4 The Simple Features standard

Geometries are the basic building blocks of vector layers. Although the Simple Features standard defines about 20 types of geometries, we will focus on the seven most commonly used types: `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON` and `GEOMETRYCOLLECTION`. A useful list of possible geometry types can be found in R's **sf** package documentation[1].

Simple feature geometries can be represented by well-known binary (WKB) and well-known text (WKT) encodings. WKB representations are usually hexadecimal strings easily readable for computers, and this is why GIS software and spatial databases use WKB to transfer and store geometry objects. WKT, on the other hand, is a human-readable text markup description of Simple Features. Both formats are exchangeable, and if we present one, we will naturally choose the WKT representation.

The foundation of each geometry type is the point. A point is simply a coordinate in two-dimensional, three-dimensional, or four-dimensional space such as shown in Figure 1.7.

```
POINT (5 2)
```

A linestring is a sequence of points with a straight line connecting the points (Figure 1.8).

```
LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)
```

A polygon is a sequence of points that form a closed, non-intersecting ring. Closed means that the first and the last point of a polygon have the same coordinates (Figure 1.9).

```
POLYGON ((1 5, 2 2, 4 1, 4 4, 1 5))
```

So far we have created geometries with only one geometric entity per feature. However, the Simple Features standard allows multiple geometries to exist within a single feature, using 'multi' versions of each geometry type, as illustrated in Figure 1.10, Figure 1.11, and Figure 1.12.

```
MULTIPOINT (5 2, 1 3, 3 4, 3 2)
MULTILINESTRING ((1 5, 4 4, 4 1, 2 2, 3 2), (1 2, 2 4))
MULTIPOLYGON (((1 5, 2 2, 4 1, 4 4, 1 5), (0 2, 1 2, 1 3, 0 3, 0 2)))
```

[1] https://r-spatial.github.io/sf/articles/sf1.html#simple-feature-geometry-types

Finally, a geometry collection can contain any combination of geometries of the other six types, such as the combination of a multipoint and linestring shown below (Figure 1.13).

```
GEOMETRYCOLLECTION (MULTIPOINT (5 2, 1 3, 3 4, 3 2),
                    LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2))
```

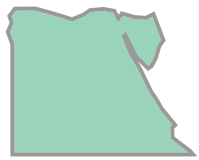
1.2.5 Geometries

Each element in the geometry column (`GeoSeries`) is a geometry object of class `shapely` (Gillies et al. 2007--). For example, here is one specific geometry selected by implicit index (Canada, the 4th element in `gdf`'s geometry column).

```
gdf.geometry.iloc[3]
```

We can also select a specific geometry based on the `'name_long'` attribute (i.e., the 1st and only element in the subset of `gdf` where the country name is equal to `Egypt`):

```
gdf[gdf['name_long'] == 'Egypt'].geometry.iloc[0]
```



The **shapely** package is compatible with the Simple Features standard (Section 1.2.4). Accordingly, seven types of geometry types are supported. The following section demonstrates creating a `shapely` geometry of each type from scratch. In the first example (a `'Point'`) we show two types of inputs to create a geometry: a list of coordinates or a `string` in the WKT format. In the examples for the remaining geometries we use the former approach.

Creating a `'Point'` geometry from a list of coordinates uses the `shapely.Point` function in the following expression (Figure 1.7).

```
point = shapely.Point([5, 2])
point
```

Figure 1.7: A `Point` geometry (created either from a `list` or WKT)

Alternatively, we can use `shapely.from_wkt` to transform a WKT string to a `shapely` geometry object. Here is an example of creating the same `'Point'` geometry from WKT (Figure 1.7).

```
point = shapely.from_wkt('POINT (5 2)')
point
```

A `'LineString'` geometry can be created based on a list of coordinate tuples or lists (Figure 1.8).

```
linestring = shapely.LineString([(1,5), (4,4), (4,1), (2,2), (3,2)])
linestring
```

Figure 1.8: A `LineString` geometry

Creation of a `'Polygon'` geometry is similar, but our first and last coordinate must be the same, to ensure that the polygon is closed. Note that in the following example, there is one list of coordinates that defines the exterior outer hull of the polygon, followed by a `list` of `lists` of coordinates that define the holes (if any) in the polygon (Figure 1.9).

```
polygon = shapely.Polygon(
    [(1,5), (2,2), (4,1), (4,4), (1,5)],  ## Exterior
    [[(2,4), (3,4), (3,3), (2,3), (2,4)]]  ## Hole(s)
)
polygon
```

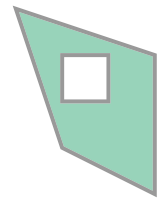


Figure 1.9: A `Polygon` geometry

A `'MultiPoint'` geometry is also created from a list of coordinate tuples (Figure 1.10), where each element represents a single point.

```
multipoint = shapely.MultiPoint([(5,2), (1,3), (3,4), (3,2)])
multipoint
```

Figure 1.10: A `MultiPoint` geometry

A `'MultiLineString'` geometry, on the other hand, has one list of coordinates for each line in the `MultiLineString` (Figure 1.11).

```
multilinestring = shapely.MultiLineString([
    [(1,5), (4,4), (4,1), (2,2), (3,2)],  ## 1st sequence
    [(1,2), (2,4)]  ## 2nd sequence, etc.
])
multilinestring
```

Figure 1.11: A `MultiLineString` geometry

A `'MultiPolygon'` geometry (Figure 1.12) is created from a `list` of `Polygon` geometries. For example, here we are creating a `'MultiPolygon'` with two parts, both without holes.

```
multipolygon = shapely.MultiPolygon([
    [[(1,5), (2,2), (4,1), (4,4), (1,5)], []],  ## Polygon 1
    [[(0,2), (1,2), (1,3), (0,3), (0,2)], []]   ## Polygon 2, etc.
])
multipolygon
```

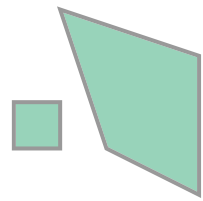


Figure 1.12: A `MultiPolygon` geometry

Since the required input has four hierarchical levels, it may be more clear to create the single-part `'Polygon'` geometries in advance, using the respective function (`shapely.Polygon`), and then pass them to `shapely.MultiPolygon` (Figure 1.12). (The same technique can be used with the other `shapely.Multi*` functions.)

```
multipolygon = shapely.MultiPolygon([
    shapely.Polygon([(1,5), (2,2), (4,1), (4,4), (1,5)]),  ## Polygon 1
    shapely.Polygon([(0,2), (1,2), (1,3), (0,3), (0,2)])   ## Polygon 2, etc.
])
multipolygon
```

And, finally, a `'GeometryCollection'` geometry is a `list` with one or more of the other six geometry types (Figure 1.13):

```
geometrycollection = shapely.GeometryCollection([multipoint, multilinestring])
geometrycollection
```

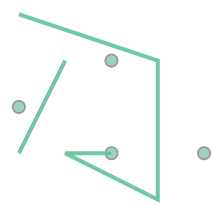


Figure 1.13: A `'GeometryCollection'` geometry

`shapely` geometries act as atomic units of vector data, meaning that there is no concept of geometry *sets*: each operation accepts individual geometry object(s) as input, and returns an individual geometry as output. (The `GeoSeries` and `GeoDataFrame` objects, defined in **geopandas**, are used to deal with sets of `shapely` geometries, collectively.) For example, the following expression calculates the difference (see Section 4.2.5) between the buffered (see Section 4.2.3) `multipolygon` (using distance of `0.2`) and itself (Figure 1.14):

```
multipolygon.buffer(0.2).difference(multipolygon)
```

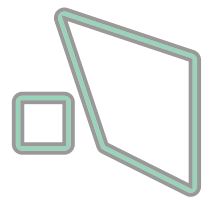


Figure 1.14: The difference between a buffered `MultiPolygon` and itself

As demonstrated in the last few figures, a `shapely` geometry object is automatically evaluated to a small image of the geometry (when using an interface capable of displaying it, such as Jupyter Notebook). To print the WKT string instead, we can use the `print` function:

```
print(linestring)
```

```
LINESTRING (1 5, 4 4, 4 1, 2 2, 3 2)
```

Finally, it is important to note that raw coordinates of `shapely` geometries are accessible through a combination of the `.coords`, `.geoms`, `.exterior`, and `.interiors` properties (depending on the geometry type). These access methods are helpful when we need to develop our own spatial operators for specific tasks. For example, the following expression returns the `list` of all coordinates of the `polygon` geometry exterior:

```
list(polygon.exterior.coords)
```

```
[(1.0, 5.0), (2.0, 2.0), (4.0, 1.0), (4.0, 4.0), (1.0, 5.0)]
```

Also see Section 4.2.8, where `.coords`, `.geoms`, and `.exterior` are used to transform a given `shapely` geometry to a different type (e.g., `'Polygon'` to `'MultiPoint'`).

1.2.6 Vector layer from scratch

In the previous sections, we started with a vector layer (`GeoDataFrame`), from an existing GeoPackage file, and 'decomposed' it to extract the geometry column

(`GeoSeries`, Section 1.2.3) and separate geometries (`shapely`, see Section 1.2.5). In this section, we will demonstrate the opposite process, constructing a `GeoDataFrame` from `shapely` geometries, combined into a `GeoSeries`. This will help you better understand the structure of a `GeoDataFrame`, and may come in handy when you need to programmatically construct simple vector layers, such as a line between two given points.

Vector layers consist of two main parts: geometries and non-geographic attributes. Figure 1.15 shows how a `GeoDataFrame` object is created—geometries come from a `GeoSeries` object (which consists of `shapely` geometries), while attributes are taken from `Series` objects.

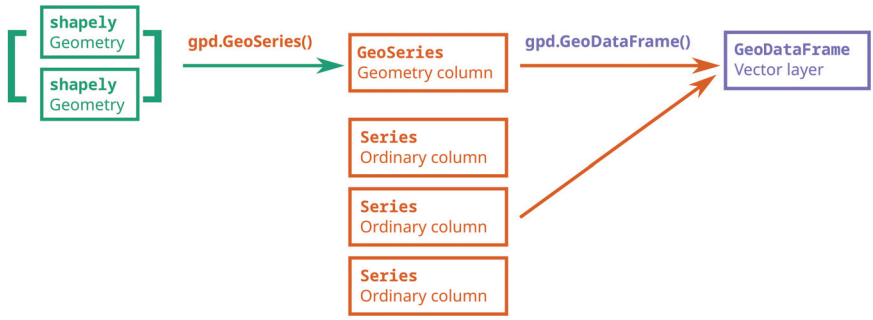


Figure 1.15: Creating a `GeoDataFrame` from scratch

The final result, a vector layer (`GeoDataFrame`) is therefore a hierarchical structure (Figure 1.16), containing the geometry column (`GeoSeries`), which in turn contains geometries (`shapely`). Each of the 'internal' components can be accessed, or 'extracted', which is sometimes necessary, as we will see later on.

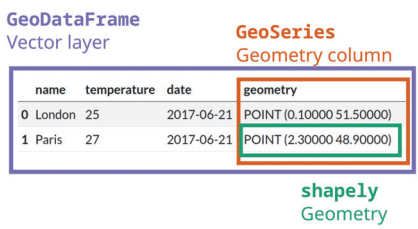


Figure 1.16: Structure of a `GeoDataFrame`

Non-geographic attributes may represent the name of the feature, and other attributes such as measured values, groups, etc. To illustrate attributes, we will represent a temperature of 25°C in London on June 21st, 2023. This example contains a geometry (the coordinates), and three attributes with

three different classes (place name, temperature, and date). Objects of class `GeoDataFrame` represent such data by combining the attributes (`Series`) with the simple feature geometry column (`GeoSeries`). First, we create a point geometry, which we know how to do from Section 1.2.5 (Figure 1.17).

```
lnd_point = shapely.Point(0.1, 51.5)
lnd_point
```

Figure 1.17: A `shapely` point representing London

Next, we create a `GeoSeries` (of length 1), containing the point and a CRS definition, in this case WGS84 (defined using its EPSG code `4326`). Also note that the `shapely` geometries go into a `list`, to illustrate that there can be more than one geometry unlike in this example.

```
lnd_geom = gpd.GeoSeries([lnd_point], crs=4326)
lnd_geom
```

```
0    POINT (0.1 51.5)
dtype: geometry
```

Next, we combine the `GeoSeries` with other attributes into a `dict`. The geometry column is a `GeoSeries`, named `geometry`. The other attributes (if any) may be defined using `list` or `Series` objects. Here, for simplicity, we use the `list` option for defining the three attributes `name`, `temperature`, and `date`. Again, note that the `list` can be of length >1, in case we are creating a layer with more than one feature (i.e., multiple rows).

```
lnd_data = {
  'name': ['London'],
  'temperature': [25],
  'date': ['2023-06-21'],
  'geometry': lnd_geom
}
```

Finally, the `dict` can be converted to a `GeoDataFrame` object, as shown in the following code.

```
lnd_layer = gpd.GeoDataFrame(lnd_data)
lnd_layer
```

	name	temperature	date	geometry
0	London	25	2023-06-21	POINT (0.1 51.5)

What just happened? First, the coordinates were used to create the simple feature geometry (`shapely`). Second, the geometry was converted into a simple feature geometry column (`GeoSeries`), with a CRS. Third, attributes were combined with `GeoSeries`. This results in an `GeoDataFrame` object, named `lnd_layer`.

To illustrate how does creating a layer with more than one feature looks like, here is an example where we create a layer with two points, London and Paris.

```
lnd_point = shapely.Point(0.1, 51.5)
paris_point = shapely.Point(2.3, 48.9)
towns_geom = gpd.GeoSeries([lnd_point, paris_point], crs=4326)
towns_data = {
  'name': ['London', 'Paris'],
  'temperature': [25, 27],
  'date': ['2013-06-21', '2013-06-21'],
  'geometry': towns_geom
}
towns_layer = gpd.GeoDataFrame(towns_data)
towns_layer
```

	name	temperature	date	geometry
0	London	25	2013-06-21	POINT (0.1 51.5)
1	Paris	27	2013-06-21	POINT (2.3 48.9)

Now, we are able to create an interactive map of the `towns_layer` object (Figure 1.18). To make the points easier to see, we are customizing a fill color and size (we elaborate on `.explore` options in Section 8.3).

```
towns_layer.explore(color='red', marker_kwds={'radius': 10})
```

A spatial (point) layer can be also created from a `DataFrame` object (package **pandas**) that contains columns with coordinates. To demonstrate, we hereby first create a `GeoSeries` object from the coordinates, and then combine it with the `DataFrame` to form a `GeoDataFrame`.

```
towns_table = pd.DataFrame({
  'name': ['London', 'Paris'],
  'temperature': [25, 27],
  'date': ['2017-06-21', '2017-06-21'],
  'x': [0.1, 2.3],
  'y': [51.5, 48.9]
})
towns_geom = gpd.points_from_xy(towns_table['x'], towns_table['y'])
towns_layer = gpd.GeoDataFrame(towns_table, geometry=towns_geom, crs=4326)
```

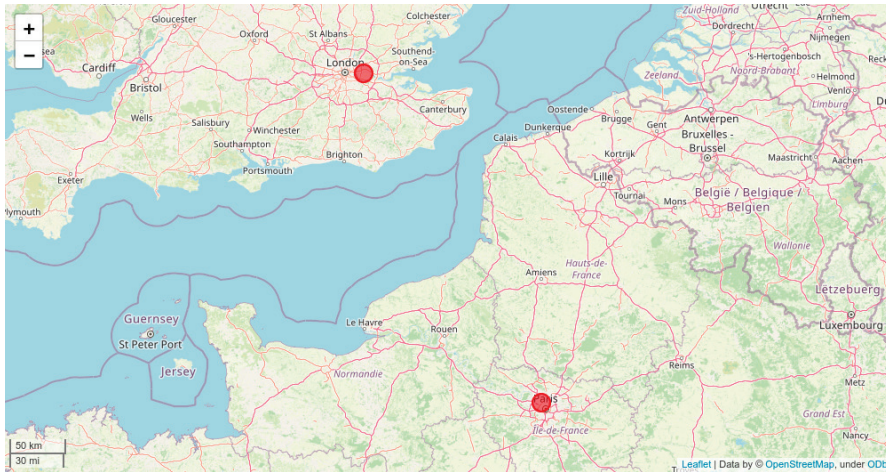


Figure 1.18: `towns_layer`, created from scratch, visualized using `.explore`

The output gives the same result as previous `towns_layer`. This approach is particularly useful when we need to read data from a CSV file, e.g., using `pd.read_csv`, and want to turn the resulting `DataFrame` into a `GeoDataFrame` (see another example in Section 3.2.3).

1.2.7 Derived numeric properties

Vector layers are characterized by two essential derived numeric properties: *length* (`.length`)—applicable to lines, and *area* (`.area`)—applicable to polygons. Area and length can be calculated for any data structures discussed above, either a `shapely` geometry, in which case the returned value is a number, or for `GeoSeries` or `DataFrame`, in which case the returned value is a numeric `Series`.

```
linestring.length
```

```
9.39834563766817
```

```
multipolygon.area
```

```
8.0
```

```
gpd.GeoSeries([point, linestring, polygon, multipolygon]).area
```

```
0    0.0
1    0.0
2    6.0
3    8.0
dtype: float64
```

Like all numeric calculations in **geopandas**, the results assume a planar CRS and are returned in its native units. This means that length and area measurements for geometries in WGS84 (`crs=4326`) are returned in decimal degrees and essentially meaningless (to see the warning, try running `gdf.area`).

To obtain meaningful length and area measurements for data in a geographic CRS, the geometries first need to be transformed to a projected CRS (see Section 6.7) applicable to the area of interest. For example, the area of Slovenia can be calculated in the UTM zone 33N CRS (`crs=32633`). The result is in m^2, the units of the UTM zone 33N CRS.

```
gdf[gdf['name_long'] == 'Slovenia'].to_crs(32633).area
```

```
150    1.910410e+10
dtype: float64
```

1.3 Raster data

The spatial raster data model represents the world with the continuous grid of cells (often also called pixels; Figure 1.19 (A)). This data model often refers to so-called regular grids, in which each cell has the same, constant size—and we will focus only on regular grids in this book. However, several other types of grids exist, including rotated, sheared, rectilinear, and curvilinear grids (see Chapter 1 of Pebesma and Bivand (2022) or Chapter 2 of Tennekes and Nowosad (2022)).

The raster data model usually consists of a raster header (or metadata) and a matrix (with rows and columns) representing equally spaced cells (often also called pixels; Figure 1.19 (A)). The raster header defines the coordinate reference system, the origin and the resolution. The origin (or starting point) is typically the coordinate of the lower-left corner of the matrix. The metadata defines the origin, and the cell size, i.e., resolution. Combined with the column and row count, the extent can also be derived. The matrix representation avoids storing explicitly the coordinates for the four corner points (in fact it only stores one coordinate, namely the origin) of each cell, as would be the case for rectangular vector polygons. This and map algebra (Section 3.3.2) makes raster processing much more efficient and faster than vector data processing. However, in contrast to vector data, the cell of one raster layer can only hold a single value. The cell values are numeric, representing either a continuous or a categorical variable (Figure 1.19 (C)).

Raster maps usually represent continuous phenomena such as elevation, temperature, population density, or spectral data. Discrete features such as soil or land-cover classes can also be represented in the raster data model. Both uses

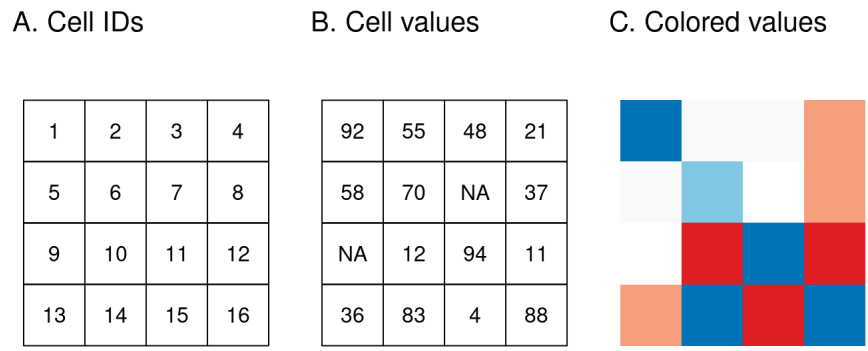


Figure 1.19: Raster data types: (A) cell IDs, (B) cell values, (C) a colored raster map

of raster datasets are illustrated in Figure 1.20, which shows how the borders of discrete features may become blurred in raster datasets. Depending on the nature of the application, vector representations of discrete features may be more suitable.

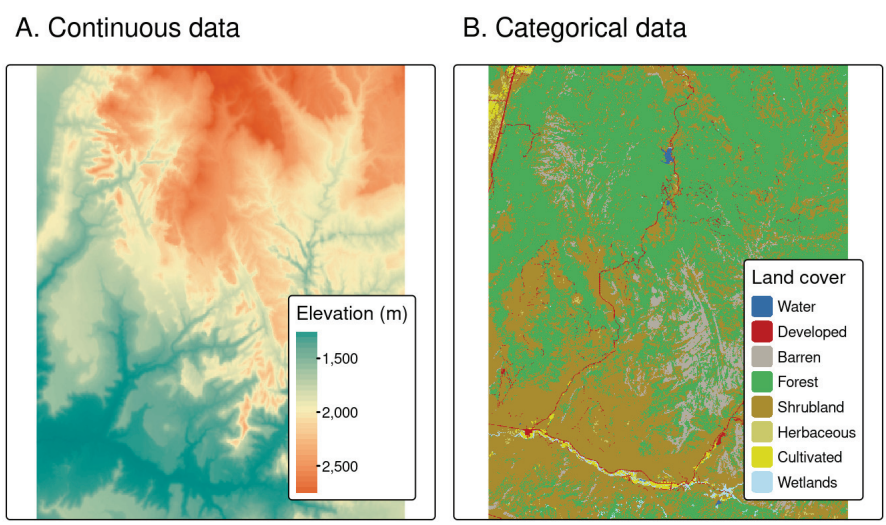


Figure 1.20: Examples of continuous and categorical rasters

As mentioned above, working with rasters in Python is less organized around one comprehensive package as compared to the case for vector layers and **geopandas**. Instead, several packages provide alternative subsets of methods for working with raster data.

The two most notable approaches for working with rasters in Python are provided by **rasterio** and **rioxarray** packages. As we will see shortly, they differ in scope and underlying data models. Specifically, **rasterio** represents rasters as **numpy** arrays associated with a separate object holding the spatial metadata. The **rioxarray** package, a wrapper of **rasterio**, however, represents rasters with **xarray** 'extended' arrays, which are an extension of **numpy** array designed to hold axis labels and attributes in the same object, together with the array of raster values. Similar approaches are provided by less well-known **xarray-spatial** and **geowombat** packages. Comparatively, **rasterio** is more well-established, but it is more low-level (which has both advantages and disadvantages).

All of the above-mentioned packages, however, are not exhaustive in the same way **geopandas** is. For example, when working with **rasterio**, more packages may be needed to accomplish common tasks such as zonal statistics (package **rasterstats**) or calculating topographic indices (package **richdem**).

In the following two sections, we introduce **rasterio**, which is the raster-related package we are going to work with through the rest of the book.

1.3.1 Using rasterio

To work with the **rasterio** package, we first need to import it. Additionally, as the raster data is stored within **numpy** arrays, we import the **numpy** package and make all its functions accessible for effective data manipulation. Finally, we import the **rasterio.plot** sub-module for its `rasterio.plot.show` function that allows for quick visualization of rasters.

```
import numpy as np
import rasterio
import rasterio.plot
```

Rasters are typically imported from existing files. When working with **rasterio**, importing a raster is actually a two-step process:

- First, we open a raster file 'connection' using `rasterio.open`
- Second, we read raster values from the connection using the `.read` method

This type of separation is analogous to basic Python functions for reading from files, such as `open` and `.readline` to read from a text file. The rationale is that we do not always want to read all information from the file into memory, which is particularly important as rasters size can be larger than RAM size. Accordingly, the second step (`.read`) is selective, meaning that the user can

fine-tune the subset of values (bands, rows/columns, resolution, etc.) that are actually being read. For example, we may want to read just one raster band rather than reading all bands.

In the first step, we pass a file path to the `rasterio.open` function to create a `DatasetReader` file connection, hereby named `src`. For this example, we use a single-band raster representing elevation in Zion National Park, stored in `srtm.tif`.

```
src = rasterio.open('data/srtm.tif')
src
```

```
<open DatasetReader name='data/srtm.tif' mode='r'>
```

To get a first impression of the raster values, we can plot the raster using the `rasterio.plot.show` function (Figure 1.21):

```
rasterio.plot.show(src);
```

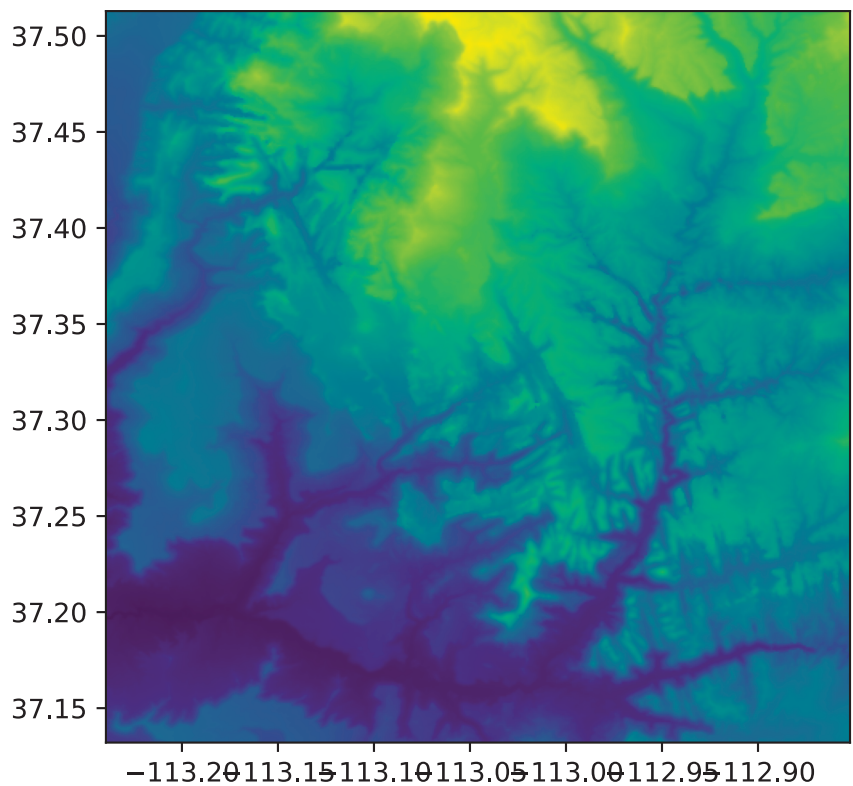


Figure 1.21: Basic plot of a raster, the data are coming from a **rasterio** file connection

The `DatasetReader` contains the raster metadata, that is, all of the information other than the raster values. Let's examine it with the `.meta` property.

```
src.meta
```

```
{'driver': 'GTiff',
 'dtype': 'uint16',
 'nodata': 65535.0,
 'width': 465,
 'height': 457,
 'count': 1,
 'crs': CRS.from_epsg(4326),
 'transform': Affine(0.0008333333332777796, 0.0, -113.23958321278403,
        0.0, -0.0008333333332777843, 37.512916763165805)}
```

Namely, it allows us to see the following properties, which we will elaborate on below, and in later chapters:

- `driver`—The raster file format (see Section 7.6.2)
- `dtype`—Data type (see Table 7.2)
- `nodata`—The value being used as 'No Data' flag (see Section 7.6.2)
- Dimensions:
  - `width`—Number of columns
  - `height`—Number of rows
  - `count`—Number of bands
- `crs`—Coordinate reference system (see Section 6.3)
- `transform`—The raster affine transformation matrix

The last item (i.e., `transform`) deserves more attention. To position a raster in geographical space, in addition to the CRS, we must specify the raster *origin* (x_{min}, y_{max}) and resolution ($delta_x$, $delta_y$). In the transformation matrix notation, assuming a regular grid, these data items are stored as follows:

```
Affine(delta_x, 0.0, x_min,
       0.0, delta_y, y_max)
```

Note that, by convention, raster y-axis origin is set to the maximum value (y_{max}) rather than the minimum, and, accordingly, the y-axis resolution ($delta_y$) is negative. In other words, since the origin is in the *top*-left corner, advancing along the y-axis is done through negative steps (downwards).

In the second step, the `.read` method of the `DatasetReader` is used to read the actual raster values. Importantly, we can read:

- All layers (as in `.read()`)
- A particular layer, passing a numeric index (as in `.read(1)`)
- A subset of layers, passing a `list` of indices (as in `.read([1,2])`)

Note that the layer indices start from `1`, contrary to the Python convention of the first index being `0`.

The object returned by `.read` is a **numpy** array (Harris et al. 2020), with either two or three dimensions:

- *Three* dimensions, when reading more than one layer (e.g., `.read()` or `.read([1,2])`). In such case, the dimensions pattern is `(layers, rows, columns)`
- *Two* dimensions, when reading one specific layer (e.g., `.read(1)`). In such case, the dimensions pattern is `(rows, columns)`

Let's read the first (and only) layer from the `srtm.tif` raster, using the file connection object `src` and the `.read` method.

```
src.read(1)
```

```
array([[1728, 1718, 1715, ..., 2654, 2674, 2685],
       [1737, 1727, 1717, ..., 2649, 2677, 2693],
       [1739, 1734, 1727, ..., 2644, 2672, 2695],
       ...,
       [1326, 1328, 1329, ..., 1777, 1778, 1775],
       [1320, 1323, 1326, ..., 1771, 1770, 1772],
       [1319, 1319, 1322, ..., 1768, 1770, 1772]], dtype=uint16)
```

The result is a two-dimensional **numpy** array where each value represents the elevation of the corresponding pixel.

The relation between a **rasterio** file connection and the derived properties is summarized in Figure 1.22. The file connection (created with `rasterio.open`) gives access to the two components of raster data: the metadata (via the `.meta` property) and the values (via the `.read` method).

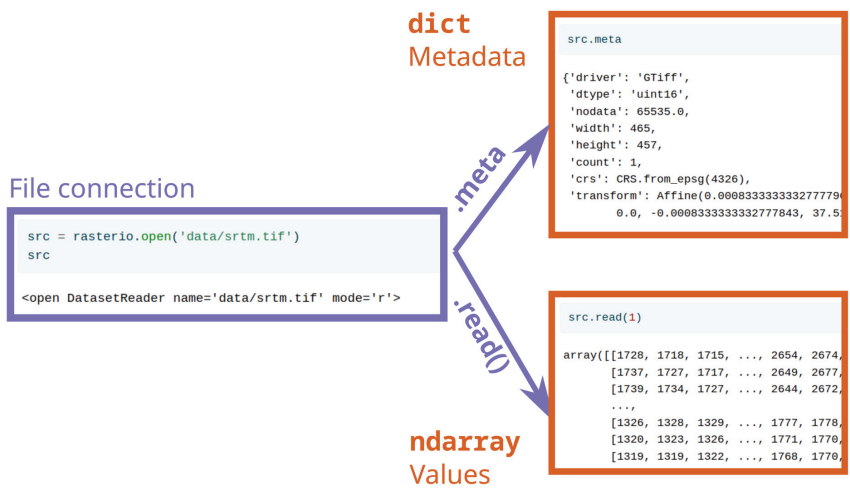


Figure 1.22: A **rasterio** file connection and its derived components, the metadata and the raster values

1.3.2 Raster from scratch

In this section, we are going to demonstrate the creation of rasters from scratch. We will construct two small rasters, `elev` and `grain`, which we will use in examples later in the book. Unlike creating a vector layer (see Section 1.2.6), creating a raster from scratch is rarely needed in practice because aligning a raster with the proper spatial extent is challenging to do programmatically ('georeferencing' tools in GIS software are a better fit for the job). Nevertheless, the examples will be helpful to become more familiar with the **rasterio** data structures.

Conceptually, a raster is an array combined with georeferencing information, whereas the latter comprises:

- A transformation matrix, containing the origin and resolution, thus linking pixel indices with coordinates in a particular coordinate system
- A CRS definition, specifying the association of that coordinate system with the surface of the earth (optional)

Therefore, to create a raster, we first need to have an array with the values, and then supplement it with the georeferencing information. Let's create the arrays `elev` and `grain`. The `elev` array is a 6×6 array with sequential values from `1` to `36`. It can be created as follows using the `np.arange` function and `.reshape` method from **numpy**.

```
elev = np.arange(1, 37, dtype=np.uint8).reshape(6, 6)
elev
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35, 36]], dtype=uint8)
```

The `grain` array represents a categorical raster with values `0`, `1`, `2`, corresponding to categories 'clay', 'silt', 'sand', respectively. We will create it from a specific arrangement of pixel values, using **numpy**'s `np.array` and `.reshape`.

```
v = [
  1, 0, 1, 2, 2, 2,
  0, 2, 0, 0, 2, 1,
  0, 2, 2, 0, 0, 2,
  0, 0, 1, 1, 1, 1,
  1, 1, 1, 2, 1, 1,
  2, 1, 2, 2, 0, 2
]
grain = np.array(v, dtype=np.uint8).reshape(6, 6)
grain
```

```
array([[1, 0, 1, 2, 2, 2],
       [0, 2, 0, 0, 2, 1],
       [0, 2, 2, 0, 0, 2],
       [0, 0, 1, 1, 1, 1],
       [1, 1, 1, 2, 1, 1],
       [2, 1, 2, 2, 0, 2]], dtype=uint8)
```

Note that, in both cases, we are using the `uint8` (unsigned integer in 8 bits, i.e., `0-255`) data type, which is sufficient to represent all possible values of the given rasters (see Table 7.2). This is the recommended approach for a minimal memory footprint.

What is missing now is the georeferencing information (see Section 1.3.1). In this case, since the rasters are arbitrary, we also set up an arbitrary transformation matrix, where:

- The origin (x_{min}, y_{max}) is at `-1.5,1.5`
- The raster resolution ($delta_x$, $delta_y$) is `0.5,-0.5`

We can add this information using `rasterio.transform.from_origin`, and specifying `west`, `north`, `xsize`, and `ysize` parameters. The resulting transformation matrix object is hereby named `new_transform`.

```
new_transform = rasterio.transform.from_origin(
    west=-1.5,
    north=1.5,
    xsize=0.5,
    ysize=0.5
)
new_transform
```

```
Affine(0.5, 0.0, -1.5,
       0.0, -0.5, 1.5)
```

Note that, confusingly, $delta_y$ (i.e., `ysize`) is defined in `rasterio.transform.from_origin` using a positive value (`0.5`), even though it is, in fact, negative (`-0.5`).

The raster can now be plotted in its coordinate system, passing the array `elev` along with the transformation matrix `new_transform` to `rasterio.plot.show` (Figure 1.23).

```
rasterio.plot.show(elev, transform=new_transform);
```

The `grain` raster can be plotted the same way, as we are going to use the same transformation matrix for it as well (Figure 1.24).

```
rasterio.plot.show(grain, transform=new_transform);
```

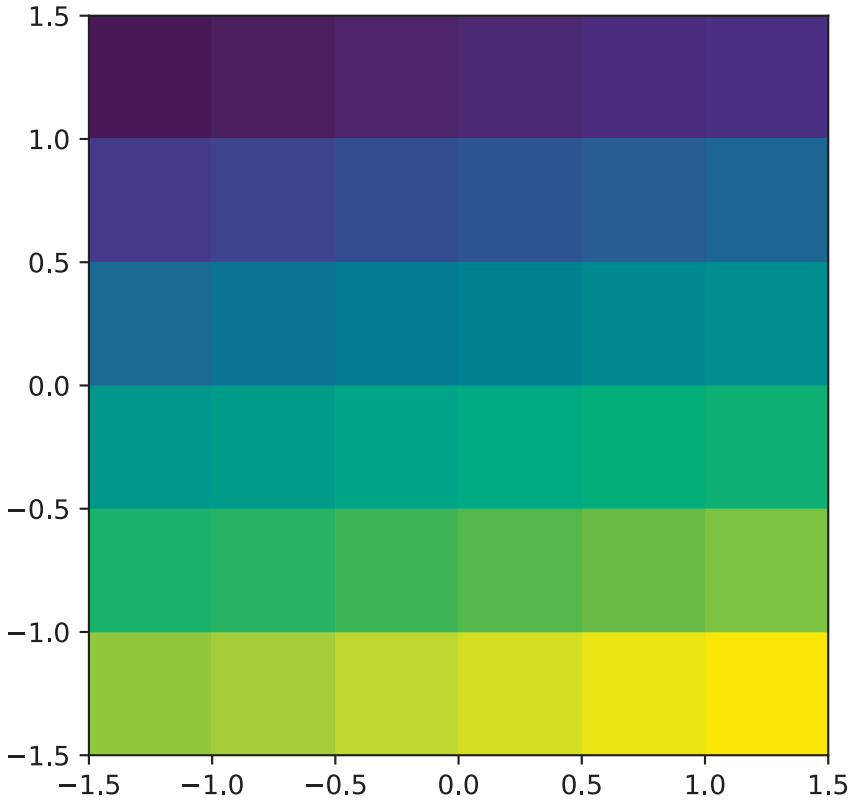


Figure 1.23: Plot of the `elev` raster, a minimal example of a continuous raster, created from scratch

At this point, we have two rasters, each composed of an array and related transformation matrix. We can work with the raster using **rasterio** by:

- Passing the transformation matrix wherever actual raster pixel coordinates are important (such as in function `rasterio.plot.show` above)
- Keeping in mind that any other layer we use in the analysis is in the same CRS

Finally, to export the raster for permanent storage, along with the spatial metadata, we need to go through the following steps:

1. Create a raster file connection (where we set the transform and the CRS, among other settings)
2. Write the array with raster values into the connection
3. Close the connection

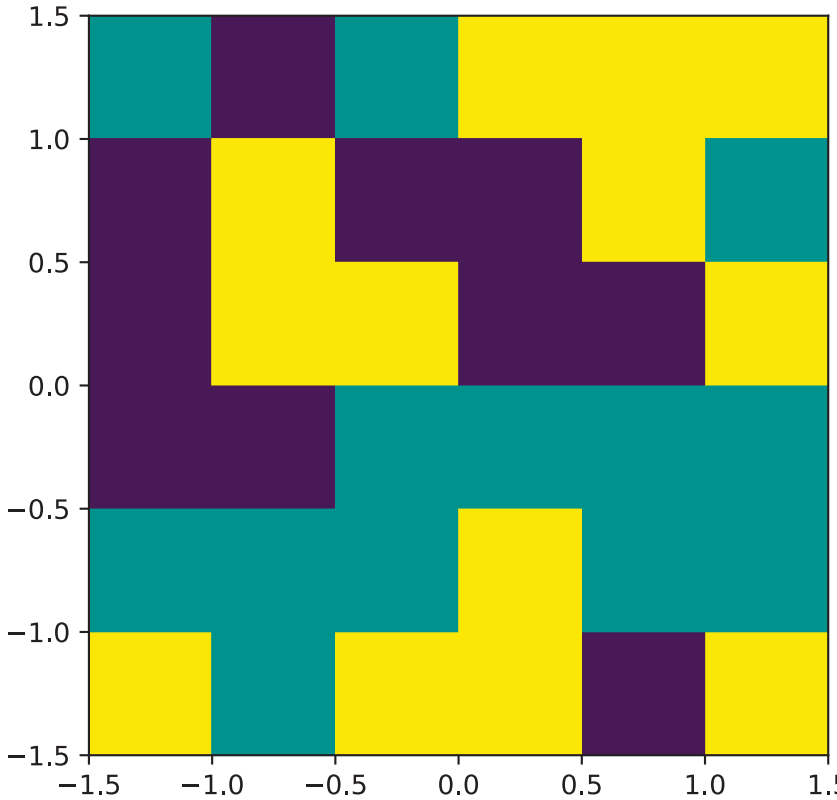


Figure 1.24: Plot of the `grain` raster, a minimal example of a categorical raster, created from scratch

Don't worry if the code below is unclear; the concepts related to writing raster data to file will be explained in Section 7.6.2. For now, for completeness, and also to use these rasters in subsequent chapters without having to re-create them from scratch, we just provide the code for exporting the `elev` and `grain` rasters into the `output` directory. In the case of `elev`, we do it as follows with the `rasterio.open`, `.write`, and `.close` functions and methods of the **rasterio** package.

```
new_dataset = rasterio.open(
    'output/elev.tif', 'w',
    driver='GTiff',
    height=elev.shape[0],
    width=elev.shape[1],
    count=1,
    dtype=elev.dtype,
```

```
    crs=4326,
    transform=new_transform
)
new_dataset.write(elev, 1)
new_dataset.close()
```

Note that the CRS we (arbitrarily) set for the `elev` raster is WGS84, defined using `crs=4326` according to the EPSG code.

Exporting the `grain` raster is done in the same way, with the only differences being the file name and the array we write into the connection.

```
new_dataset = rasterio.open(
    'output/grain.tif', 'w',
    driver='GTiff',
    height=grain.shape[0],
    width=grain.shape[1],
    count=1,
    dtype=grain.dtype,
    crs=4326,
    transform=new_transform
)
new_dataset.write(grain, 1)
new_dataset.close()
```

As a result, the files `elev.tif` and `grain.tif` are written into the `output` directory. We are going to use these small raster files later on in the examples (for example, Section 2.3.1).

Note that the transform matrices and dimensions of `elev` and `grain` are identical. This means that the rasters are overlapping, and can be combined into one two-band raster, processed in raster algebra operations (Section 3.3.2), etc.

1.4 Coordinate Reference Systems

Vector and raster spatial data types share concepts intrinsic to spatial data. Perhaps the most fundamental of these is the Coordinate Reference System (CRS), which defines how the spatial elements of the data relate to the surface of the Earth (or other bodies). CRSs are either geographic or projected, as introduced at the beginning of this chapter (Section 1.2). This section explains each type, laying the foundations for Chapter 6, which provides a deep dive into setting, transforming, and querying CRSs.

1.4.1 Geographic coordinate systems

Geographic coordinate systems identify any location on the Earth's surface using two values—longitude and latitude (see left panel of Figure 1.26). Longitude is a location in the East-West direction in angular distance from the Prime Meridian plane, while latitude is an angular distance North or South of the equatorial plane. Distances in geographic CRSs are therefore not measured in meters. This has important consequences, as demonstrated in Chapter 6.

A spherical or ellipsoidal surface represents the surface of the Earth in geographic coordinate systems. Spherical models assume that the Earth is a perfect sphere of a given radius—they have the advantage of simplicity, but, at the same time, they are inaccurate: the Earth is not a sphere! Ellipsoidal models are defined by two parameters: the equatorial radius and the polar radius. These are suitable because the Earth is compressed: the equatorial radius is around 11.5 km longer than the polar radius. The Earth is not an ellipsoid either, but it is a better approximation than a sphere.

Ellipsoids are part of a broader component of CRSs: the datum. It contains information on what ellipsoid to use and the precise relationship between the Cartesian coordinates and location on the Earth's surface. There are two types of datum—geocentric (such as WGS84) and local (such as NAD83). You can see examples of these two types of datums in Figure 1.25. Black lines represent a geocentric datum, whose center is located in the Earth's center of gravity and is not optimized for a specific location. In a local datum, shown as a purple dashed line, the ellipsoidal surface is shifted to align with the surface at a particular location. These allow local variations on Earth's surface, such as large mountain ranges, to be accounted for in a local CRS. This can be seen in Figure 1.25, where the local datum is fitted to the area of Philippines, but is misaligned with most of the rest of the planet's surface. Both datums in Figure 1.25 are put on top of a geoid—a model of global mean sea level.

1.4.2 Projected coordinate reference systems

All projected CRSs are based on a geographic CRS, described in the previous section, and rely on map projections to convert the three-dimensional surface of the Earth into Easting and Northing (x and y) values in a projected CRS. Projected CRSs are based on Cartesian coordinates on an implicitly flat surface (see right panel of Figure 1.26). They have an origin, x and y axes, and a linear unit of measurement such as meters.

This transition cannot be done without adding some deformations. Therefore, some properties of the Earth's surface are distorted in this process, such as area, direction, distance, and shape. A projected coordinate system can preserve only one or two of those properties. Projections are often named based on a property they preserve: equal-area preserves area, azimuthal preserves direction, equidistant preserves distance, and conformal preserves local shape.

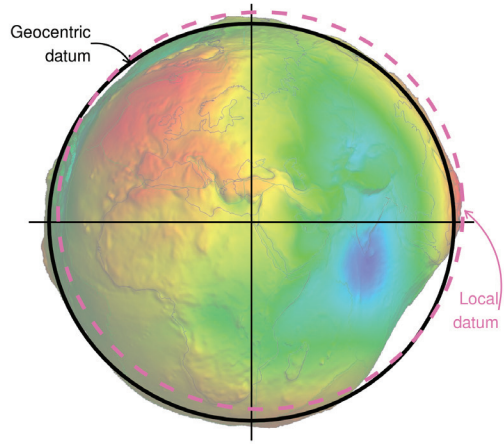


Figure 1.25: Geocentric and local geodetic datums shown on top of a geoid (in false color and the vertical exaggeration by 10,000 scale factor). Image of the geoid is adapted from the work of Ince et al. (2019).

There are three main groups of projection types: conic, cylindrical, and planar (azimuthal). In a conic projection, the Earth's surface is projected onto a cone along a single line of tangency or two lines of tangency. Distortions are minimized along the tangency lines and rise with the distance from those lines in this projection. Therefore, it is best suited for maps of mid-latitude areas. A cylindrical projection maps the surface onto a cylinder. This projection could also be created by touching the Earth's surface along a single line of tangency or two lines of tangency. Cylindrical projections are used most often when mapping the entire world. A planar projection projects data onto a flat surface touching the globe at a point or along a line of tangency. It is typically used in mapping polar regions.

1.4.3 CRS in Python

Like most open-source geospatial software, the **geopandas** and **rasterio** packages use the PROJ software for CRS definition and calculations. The **pyproj** package is a low-level interface to PROJ. Using its functions, such as `get_codes` and `from_epsg`, we can examine the list of projections supported by PROJ.

```
import pyproj
epsg_codes = pyproj.get_codes('EPSG', 'CRS')  ## Supported EPSG codes
epsg_codes[:5]  ## Print first five supported EPSG codes
```

```
['10150', '10151', '10156', '10157', '10158']
```

```
pyproj.CRS.from_epsg(4326)  ## Printout of WGS84 CRS (EPSG:4326)
```

```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

A quick summary of different projections, their types, properties, and suitability can be found at https://www.geo-projections.com/. We will expand on CRSs and explain how to project from one CRS to another in Chapter 6. But, for now, it is sufficient to know:

- That coordinate systems are a key component of geographic objects
- Knowing which CRS your data is in, and whether it is in geographic (lon/lat) or projected (typically meters), is important and has consequences for how Python handles spatial and geometry operations
- CRSs of **geopandas** (vector layer or geometry column) and **rasterio** (raster) objects can be queried with the `.crs` property

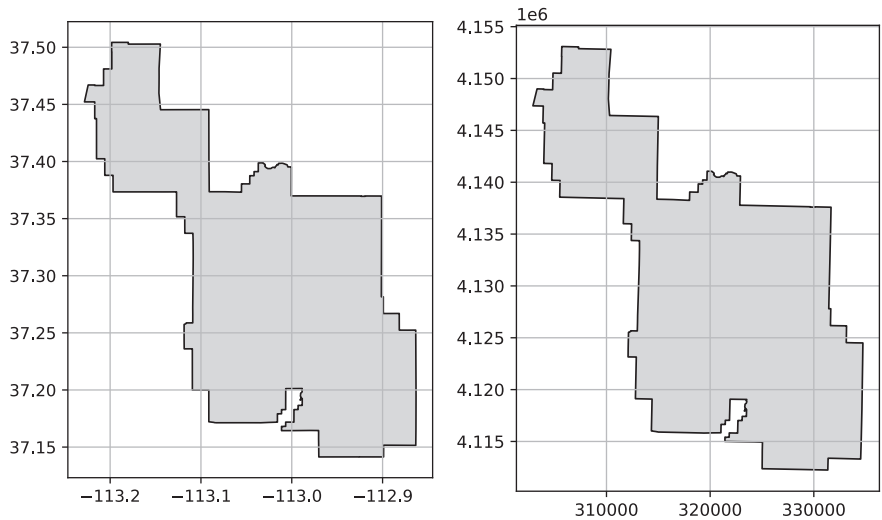
Here is a demonstration of the last bullet point, where we import a vector layer and figure out its CRS (in this case, a projected CRS, namely UTM Zone 12) using the `.crs` property.

```
zion = gpd.read_file('data/zion.gpkg')
zion.crs
```

```
<Bound CRS: PROJCS["UTM Zone 12, Northern Hemisphere",GEOGCS[" ...>
Name: UTM Zone 12, Northern Hemisphere
Axis Info [cartesian]:
- [east]: Easting (Meter)
- [north]: Northing (Meter)
Area of Use:
- undefined
Coordinate Operation:
- name: Transformation from GRS 1980(IUGG, 1980) to WGS84
- method: Position Vector transformation (geog2D domain)
Datum: unknown
- Ellipsoid: GRS80
- Prime Meridian: Greenwich
Source CRS: UTM Zone 12, Northern Hemisphere
```

We can also illustrate the difference between a geographic and a projected CRS by plotting the `zion` data in both CRSs (Figure 1.26). Note that we are using the `.grid` method of **matplotlib** to draw grid lines on top of the plot.

```
# WGS84
zion.to_crs(4326).plot(edgecolor='black', color='lightgrey').grid()
# NAD83 / UTM zone 12N
zion.plot(edgecolor='black', color='lightgrey').grid();
```



(a) Geographic (WGS84)

(b) Projected (NAD83 / UTM zone 12N)

Figure 1.26: Examples of Coordinate Reference Systems (CRS) for a vector layer

We are going to elaborate on reprojection from one CRS to another (`.to_crs` in the above code section) in Chapter 6.

1.5 Units

An important feature of CRSs is that they contain information about spatial units. Clearly, it is vital to know whether a house's measurements are in feet or meters, and the same applies to maps. It is a good cartographic practice to add a scale bar or some other distance indicator onto maps to demonstrate the relationship between distances on the page or screen and distances on the ground. Likewise, it is important for the user to be aware of the units in which

the geometry coordinates are, to ensure that subsequent calculations are done in the right context.

Python spatial data structures in **geopandas** and **rasterio** do not natively support the concept of measurement units. The coordinates of a vector layer or a raster are plain numbers, referring to an arbitrary plane. For example, according to the `.transform` matrix of `srtm.tif` we can see that the raster resolution is `0.000833` and that its CRS is WGS84 (EPSG: `4326`):

```
src.meta
```

```
{'driver': 'GTiff',
 'dtype': 'uint16',
 'nodata': 65535.0,
 'width': 465,
 'height': 457,
 'count': 1,
 'crs': CRS.from_epsg(4326),
 'transform': Affine(0.0008333333333332777796, 0.0, -113.23958321278403,
        0.0, -0.0008333333333332777843, 37.512916763165805)}
```

You may already know that the units of the WGS84 coordinate system (EPSG:4326) are decimal degrees. However, that information is not accounted for in any numeric calculation, meaning that operations such as buffers can be returned in units of degrees, which is not appropriate in most cases.

Consequently, you should always be aware of the CRS of your datasets and the units they use. Typically, these are decimal degrees, in a geographic CRS, or m, in a projected CRS, although there are exceptions. Geometric calculations such as length, area, or distance, return plain numbers in the same units of the CRS (such as m or m^2). It is up to the user to determine which units the result is given in, and treat the result accordingly. For example, if the area output was in m^2 and we need the result in km^2, then we need to divide the result by 1000^2.

2

Attribute data operations

Prerequisites

This chapter requires importing the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import geopandas as gpd
import rasterio
```

It also relies on the following data files:

```
world = gpd.read_file('data/world.gpkg')
src_elev = rasterio.open('output/elev.tif')
src_grain = rasterio.open('output/grain.tif')
src_multi_rast = rasterio.open('data/landsat.tif')
```

2.1 Introduction

Attribute data is non-spatial information associated with geographic (geometry) data. A bus stop provides a simple example: its position would typically be represented by latitude and longitude coordinates (geometry data), in addition to its name. A bus stop in London, for example, has coordinates of `-0.098` degree longitude and `51.495` degree latitude which can be represented as `POINT (-0.098 51.495)` using the Simple Feature representation described in Chapter 1. Attributes, such as the name of the bus stop, are the topic of this chapter.

Another example of an attribute is the elevation value for a specific grid cell in raster data. Unlike the vector data model, the raster data model stores the coordinate of the grid cell indirectly, meaning the distinction between attribute and spatial information is less clear. Think of a pixel in the 3rd row and the 4th column of a raster matrix: its spatial location is defined by its index in the

DOI: 10.1201/9781003379911-2

matrix. In this case, we need to move four cells in the x direction (typically east/right on maps) and three cells in the y direction (typically south/down) from the origin. The raster's resolution is also important as it defines the distance for each x- and y-step. The resolution and the origin are stored in the raster's metadata (header), which is a vital component of raster datasets which specifies how pixels relate to geographic coordinates (see also Chapter 3).

This chapter teaches how to manipulate geographic objects based on attributes such as the names of bus stops in a vector dataset and elevations of pixels in a raster dataset. For vector data, this means techniques such as subsetting and aggregation (see Section 2.2.1 and Section 2.2.2). Moreover, Section 2.2.3 and Section 2.2.4 demonstrate how to join data onto simple feature objects using a shared ID and how to create new variables, respectively. Each of these operations has a spatial equivalent: `[` operator for subsetting a `(Geo)DataFrame` using a boolean `Series`, for example, is applicable both for subsetting objects based on their attribute and spatial relations derived using methods such as `.intersects`; you can also join attributes in two geographic datasets using spatial joins. This is good news: skills developed in this chapter are cross-transferable. Chapter 3 extends the methods presented here to the spatial world.

After a deep dive into various types of vector attribute operations in the next section, raster attribute data operations are covered in Section 2.3.1, which demonstrates extracting cell values from one or more layers (raster subsetting). Section 2.3.2 provides an overview of 'global' raster operations which can be used to summarize entire raster datasets.

2.2 Vector attribute manipulation

As mentioned in Section 1.2.2, vector layers (`GeoDataFrame`, from package **geopandas**) are basically extended tables (`DataFrame` from package **pandas**), the only differences being the geometry column and class. Therefore, all ordinary table-related operations from package **pandas** are supported for **geopandas** vector layers as well, as shown below.

2.2.1 Vector attribute subsetting

pandas supports several subsetting interfaces, though the most recommended ones are `.loc`, which uses **pandas** indices, and `.iloc`, which uses (implicit) **numpy**-style numeric indices.

In both cases, the method is followed by square brackets, and two indices, separated by a comma. Each index can be:

- A specific value, as in `1`
- A `list`, as in `[0,2,4]`
- A slice, as in `0:3`
- `:`—indicating 'all' indices, as in `[:]`

An exception to this guideline is selecting columns using a list, which we do using shorter notation, as in `df[['a','b']]`, instead of `df.loc[:, ['a','b']]`, to select columns `'a'` and `'b'` from `df`.

Here are few examples of subsetting the `GeoDataFrame` of world countries (Figure 1.2). First, we are subsetting rows by position. In the first example, we are using `[0:3,:]`, meaning 'rows 1,2,3, all columns'. Keep in mind that indices in Python start from 0, and slices are inclusive of the start and exclusive of the end; therefore, `0:3` means indices `0`, `1`, `2`, i.e., first three rows in this example.

```
world.iloc[0:3, :]
```

	iso_a2	name_long	...	gdpPercap	geometry
0	FJ	Fiji	...	8222.253784	MULTIPOLYGON (((-180 -16.55522,...
1	TZ	Tanzania	...	2402.099404	MULTIPOLYGON (((33.90371 -0.95,...
2	EH	Western Sahara	...	NaN	MULTIPOLYGON (((-8.66559 27.656...

Subsetting columns by position requires specifying that we want to keep all of the rows (`:`) and then the indexes of the columns we want to keep.

```
world.iloc[:, 0:3]
```

	iso_a2	name_long	continent
0	FJ	Fiji	Oceania
1	TZ	Tanzania	Africa
2	EH	Western Sahara	Africa
...	...	...	...
174	XK	Kosovo	Europe
175	TT	Trinidad and Tobago	North America
176	SS	South Sudan	Africa

To subset rows and columns by position we need to specify both row and column indices, separated by a comma.

```
world.iloc[0:3, 0:3]
```

	iso_a2	name_long	continent
0	FJ	Fiji	Oceania
1	TZ	Tanzania	Africa
2	EH	Western Sahara	Africa

Subsetting columns by name is not done with the `.iloc` method, but instead requires specifying the column names in `.loc`, or directly in a double square bracket `[[` notation.

```
world[['name_long', 'geometry']]
```

	name_long	geometry
0	Fiji	MULTIPOLYGON (((-180 -16.55522,...
1	Tanzania	MULTIPOLYGON (((33.90371 -0.95,...
2	Western Sahara	MULTIPOLYGON (((-8.66559 27.656...
...	...	...
174	Kosovo	MULTIPOLYGON (((20.59025 41.855...
175	Trinidad and Tobago	MULTIPOLYGON (((-61.68 10.76, -...
176	South Sudan	MULTIPOLYGON (((30.83385 3.5091...

To select many successive columns, we can use the `:` (slice) notation, as in `world.loc[:, 'name_long':'pop']`, which selects all columns from `name_long` to `pop` (inclusive).

```
world.loc[:, 'name_long':'pop']
```

	name_long	continent	...	area_km2	pop
0	Fiji	Oceania	...	19289.970733	885806.0
1	Tanzania	Africa	...	932745.792357	52234869.0
2	Western Sahara	Africa	...	96270.601041	NaN
...	...	...	...	...	...
174	Kosovo	Europe	...	11230.261672	1821800.0
175	Trinidad and Tobago	North America	...	7737.809855	1354493.0
176	South Sudan	Africa	...	624909.099086	11530971.0

Removing rows or columns is done using the `.drop` method. We can remove specific rows by specifying their ids, e.g., dropping rows 2, 3, and 5 in the following example.

```
world.drop([2, 3, 5])
```

	iso_a2	name_long	...	gdpPercap	geometry
0	FJ	Fiji	...	8222.253784	MULTIPOLYGON (((-180 -16.55522,...
1	TZ	Tanzania	...	2402.099404	MULTIPOLYGON (((33.90371 -0.95,...
4	US	United States	...	51921.984639	MULTIPOLYGON (((-171.73166 63.7...
...	...	...	...	...	...
174	XK	Kosovo	...	8698.291559	MULTIPOLYGON (((20.59025 41.855...
175	TT	Trinidad and Tobago	...	31181.821196	MULTIPOLYGON (((-61.68 10.76, -...

	iso_a2	name_long	...	gdpPercap	geometry
176	SS	South Sudan	...	1935.879400	MULTIPOLYGON (((30.83385 3.5091...

To remove specific columns we need to add an extra argument, `axis=1` (i.e., columns).

```
world.drop(['name_long', 'continent'], axis=1)
```

	iso_a2	region_un	...	gdpPercap	geometry
0	FJ	Oceania	...	8222.253784	MULTIPOLYGON (((-180 -16.55522,...
1	TZ	Africa	...	2402.099404	MULTIPOLYGON (((33.90371 -0.95,...
2	EH	Africa	...	NaN	MULTIPOLYGON (((-8.66559 27.656...
...	...	...	...	...	...
174	XK	Europe	...	8698.291559	MULTIPOLYGON (((20.59025 41.855...
175	TT	Americas	...	31181.821196	MULTIPOLYGON (((-61.68 10.76, -...
176	SS	Africa	...	1935.879400	MULTIPOLYGON (((30.83385 3.5091...

We can also rename columns using the `.rename` method, in which we pass a dictionary with items of the form `old_name:new_name` to the `columns` argument.

```
world[['name_long', 'pop']].rename(columns={'pop': 'population'})
```

	name_long	population
0	Fiji	885806.0
1	Tanzania	52234869.0
2	Western Sahara	NaN
...	...	...
174	Kosovo	1821800.0
175	Trinidad and Tobago	1354493.0
176	South Sudan	11530971.0

The standard **numpy** comparison operators (Table 2.1) can be used in boolean subsetting with **pandas**/**geopandas**.

Table 2.1: Comparison operators that return boolean values (`True`/`False`).

Symbol	Name
==	Equal to
!=	Not equal to
>, <	Greater/Less than
>=, <=	Greater/Less than or equal
&, |, ~	Logical operators: And, Or, Not

The following example demonstrates logical vectors for subsetting by creating a new `GeoDataFrame` object called `small_countries` that contains only those countries and other territories from the `world` object whose surface area is smaller than 10,000 km^2. The first step is to create a logical vector (a `Series` object) that is `True` for countries with an area smaller than 10,000 km^2 and `False` otherwise. Then, we use this vector to subset the `world` dataset, which returns a new `GeoDataFrame` object containing only the small countries.

```
idx_small = world['area_km2'] < 10000  ## a logical 'Series'
small_countries = world[idx_small]
small_countries
```

	iso_a2	name_long	...	gdpPercap	geometry
45	PR	Puerto Rico	...	35066.046376	MULTIPOLYGON (((-66.28243 18.51...
79	PS	Palestine	...	4319.528283	MULTIPOLYGON (((35.39756 31.489...
89	VU	Vanuatu	...	2892.341604	MULTIPOLYGON (((166.79316 -15.6...
...	...	...	...	...	...
160	None	Northern Cyprus	...	NaN	MULTIPOLYGON (((32.73178 35.140...
161	CY	Cyprus	...	29786.365653	MULTIPOLYGON (((32.73178 35.140...
175	TT	Trinidad and Tobago	...	31181.821196	MULTIPOLYGON (((-61.68 10.76, -...

A more concise command, which omits the intermediary object by combining the two steps into one, generates the same result.

```
small_countries = world[world['area_km2'] < 10000]
small_countries
```

	iso_a2	name_long	...	gdpPercap	geometry
45	PR	Puerto Rico	...	35066.046376	MULTIPOLYGON (((-66.28243 18.51...
79	PS	Palestine	...	4319.528283	MULTIPOLYGON (((35.39756 31.489...
89	VU	Vanuatu	...	2892.341604	MULTIPOLYGON (((166.79316 -15.6...
...	...	...	...	...	...
160	None	Northern Cyprus	...	NaN	MULTIPOLYGON (((32.73178 35.140...
161	CY	Cyprus	...	29786.365653	MULTIPOLYGON (((32.73178 35.140...
175	TT	Trinidad and Tobago	...	31181.821196	MULTIPOLYGON (((-61.68 10.76, -...

We can also combine indexes using logical operators, such as `&` (and), `|` (or), and `~` (not).

```
idx_small = world['area_km2'] < 10000
idx_asia = world['continent'] == 'Asia'
world.loc[idx_small & idx_asia, ['name_long', 'continent', 'area_km2']]
```

	name_long	continent	area_km2
79	Palestine	Asia	5037.103826
160	Northern Cyprus	Asia	3786.364506
161	Cyprus	Asia	6207.006191

The various methods shown above can be chained for any combination with several subsetting steps. For example, the following code selects only countries from Asia, keeps only the `name_long` and `continent` columns, and then selects the first five rows.

```
world[world['continent'] == 'Asia']  \
    .loc[:, ['name_long', 'continent']]  \
    .iloc[0:5, :]
```

	name_long	continent
5	Kazakhstan	Asia
6	Uzbekistan	Asia
8	Indonesia	Asia
24	Timor-Leste	Asia
76	Israel	Asia

Logical operators `&`, `|`, and `~` (Table 2.1) can be used to combine multiple conditions. For example, here are all countries in North America or South America. Keep in mind that the parentheses around each condition (here, and in analogous cases using other operators) are crucial; otherwise, due to Python's precedence rules[1], the `|` operator is executed before `==` and we get an error.

```
world[
        (world['continent'] == 'North America') |
        (world['continent'] ==  'South America')
    ]  \
    .loc[:, ['name_long', 'continent']]
```

[1] https://docs.python.org/3/reference/expressions.html#operator-precedence

	name_long	continent
3	Canada	North America
4	United States	North America
9	Argentina	South America
...	...	...
47	Cuba	North America
156	Paraguay	South America
175	Trinidad and Tobago	North America

However, specifically, expressions combining multiple comparisons with `==` combined with `|` can be replaced with the `.isin` method and a `list` of values to compare with. The advantage of `.isin` is more concise and easy to manage code, especially when the number of comparisons is large. For example, the following expression gives the same result as above.

```
world[world['continent'].isin(['North America', 'South America'])]  \
    .loc[:, ['name_long', 'continent']]
```

	name_long	continent
3	Canada	North America
4	United States	North America
9	Argentina	South America
...	...	...
47	Cuba	North America
156	Paraguay	South America
175	Trinidad and Tobago	North America

2.2.2 Vector attribute aggregation

Aggregation involves summarizing data based on one or more *grouping variables* (typically values in a column; geographic aggregation is covered in Section 3.2.5). A classic example of this attribute-based aggregation is calculating the number of people per continent based on country-level data (one row per country). The `world` dataset contains the necessary ingredients: the columns `pop` and `continent`, the target variable and the grouping variable, respectively. The aim is to find the `sum()` of country populations for each continent, resulting in a smaller table or vector layer (of continents). Since aggregation is a form of data reduction, it can be a useful early step when working with large datasets.

Attribute-based aggregation can be achieved using a combination of `.groupby` and `.sum` (package **pandas**), where the former groups the data by the grouping variable(s) and the latter calculates the sum of the specified column(s). The `.reset_index` method moves the grouping variable into an ordinary column, rather than an index (the default), which is something we typically want to do.

```
world_agg1 = world.groupby('continent')[['pop']].sum().reset_index()
world_agg1
```

	continent	pop
0	Africa	1.154947e+09
1	Antarctica	0.000000e+00
2	Asia	4.311408e+09
...	...	...
5	Oceania	3.775783e+07
6	Seven seas (open ocean)	0.000000e+00
7	South America	4.120608e+08

The result, in this case, is a (non-spatial) table with eight rows, one per unique value in `continent`, and two columns reporting the name and population of each continent.

If we want to include the geometry in the aggregation result, we can use the `.dissolve` method. That way, in addition to the summed population, we also get the associated geometry per continent, i.e., the union of all countries. Note that we use the `by` parameter to choose which column(s) are used for grouping, and the `aggfunc` parameter to choose the aggregation function for non-geometry columns. Again, note that the `.reset_index` method is used (here, and elsewhere in the book) to turn **pandas** and **geopandas** row *indices*, which are automatically created for grouping variables in grouping operations such as `.dissolve`, 'back' into ordinary columns, which are more appropriate in the scope of this book.

```
world_agg2 = world[['continent', 'pop', 'geometry']] \
    .dissolve(by='continent', aggfunc='sum') \
    .reset_index()
world_agg2
```

	continent	geometry	pop
0	Africa	MULTIPOLYGON (((-11.43878 6.785...	1.154947e+09
1	Antarctica	MULTIPOLYGON (((-61.13898 -79.9...	0.000000e+00
2	Asia	MULTIPOLYGON (((48.67923 14.003...	4.311408e+09
...	...	...	...
5	Oceania	MULTIPOLYGON (((147.91405 -43.2...	3.775783e+07
6	Seven seas (open ocean)	POLYGON ((68.935 -48.625, 68.86...	0.000000e+00
7	South America	MULTIPOLYGON (((-68.63999 -55.5...	4.120608e+08

In this case, the resulting `world_agg2` object is a `GeoDataFrame` containing 8 features representing the continents of the world that we can plot (Figure 2.1). The `plt.subplots` function is hereby used to control plot dimensions (to make the plot wider and narrower) (see Section 8.2.2).

```
fig, ax = plt.subplots(figsize=(6, 3))
world_agg2.plot(column='pop', edgecolor='black', legend=True, ax=ax);
```

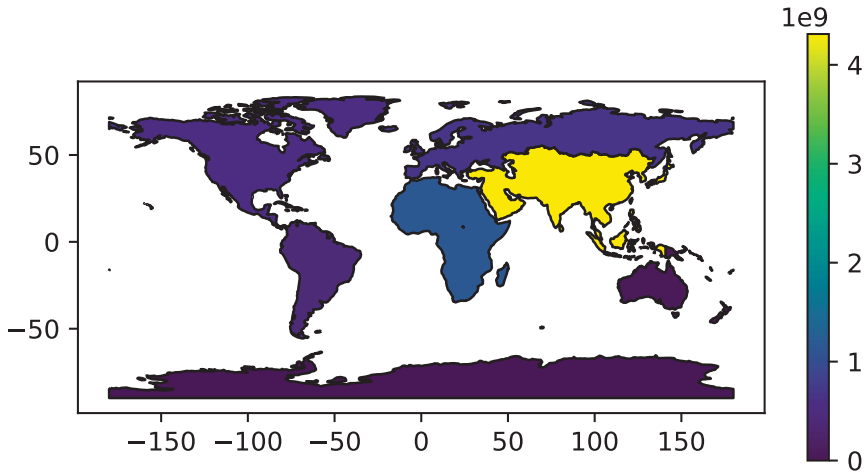


Figure 2.1: Continents with summed population

Other options for the `aggfunc` parameter in `.dissolve` include `'first'`, `'last'`, `'min'`, `'max'`, `'sum'`, `'mean'`, `'median'`. Additionally, we can pass custom functions here.

As a more complex example, the following code shows how we can calculate the total population, area, and count of countries, per continent. It is done by passing a dictionary to the `aggfunc` parameter, where the keys are the column names and the values are the aggregation functions. The result is a `GeoDataFrame` object with 8 rows (one per continent) and 4 columns (one for the continent name and one for each of the three aggregated attributes). The `rename` method is used to rename the `'name_long'` column into `'n'`, as it now expresses the count of names (i.e., the number of countries) rather than their names.

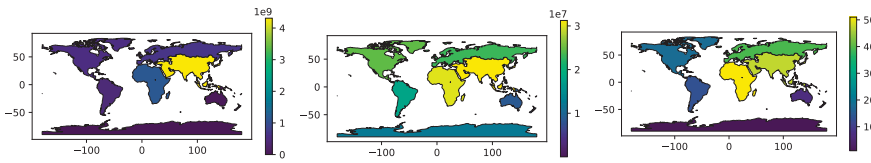
```
world_agg3 = world.dissolve(
    by='continent',
    aggfunc={
        'name_long': 'count',
        'pop': 'sum',
        'area_km2': 'sum'
    }).rename(columns={'name_long': 'n'}).reset_index()
world_agg3
```

	continent	geometry	n	pop	area_km2
0	Africa	MULTIPOLYGON (((-11.43878 6.785...	51	1.154947e+09	2.994620e+07
1	Antarctica	MULTIPOLYGON (((-61.13898 -79.9...	1	0.000000e+00	1.233596e+07

	continent	geometry	n	pop	area_km2
2	Asia	MULTIPOLYGON (((48.67923 14.003...	47	4.311408e+09	3.125246e+07
...	...	...	...	...	...
5	Oceania	MULTIPOLYGON (((147.91405 -43.2...	7	3.775783e+07	8.504489e+06
6	Seven seas (open ocean)	POLYGON ((68.935 -48.625, 68.86...	1	0.000000e+00	1.160257e+04
7	South America	MULTIPOLYGON (((-68.63999 -55.5...	13	4.120608e+08	1.776259e+07

Figure 2.2 visualizes the three aggregated attributes of our resulting layer `world_agg3`.

```
# Summed population
fig, ax = plt.subplots(figsize=(5, 2.5))
world_agg3.plot(column='pop', edgecolor='black', legend=True, ax=ax);
# Summed area
fig, ax = plt.subplots(figsize=(5, 2.5))
world_agg3.plot(column='area_km2', edgecolor='black', legend=True, ax=ax);
# Count of countries
fig, ax = plt.subplots(figsize=(5, 2.5))
world_agg3.plot(column='n', edgecolor='black', legend=True, ax=ax);
```



(a) Summed population (b) Summed area (c) Count of countries

Figure 2.2: Continent's properties, calculated using spatial aggregation using different functions

There are several other table-related operations that are possible, such as creating new columns or sorting the values. In the following code example, given the `world_agg3` continent summary (Figure 2.2), we:

- drop the geometry column,
- calculate population density of each continent,
- arrange continents by the number of countries each contains, and
- keep only the 3 most country-rich continents.

```
world_agg4 = world_agg3.drop(columns=['geometry'])
world_agg4['density'] = world_agg4['pop'] / world_agg4['area_km2']
world_agg4 = world_agg4.sort_values(by='n', ascending=False)
world_agg4 = world_agg4.head(3)
world_agg4
```

	continent	n	pop	area_km2	density
0	Africa	51	1.154947e+09	2.994620e+07	38.567388
2	Asia	47	4.311408e+09	3.125246e+07	137.954201
3	Europe	39	6.690363e+08	2.306522e+07	29.006283

2.2.3 Vector attribute joining

Combining data from different sources is a common task in data preparation. Joins do this by combining tables based on a shared 'key' variable. **pandas** has a function named `pd.merge` for joining `(Geo)DataFrames` based on common column(s) that follows conventions used in the database language SQL (Grolemund and Wickham 2016). The `pd.merge` result can be either a `DataFrame` or a `GeoDataFrame` object, depending on the inputs.

A common type of attribute join on spatial data is to join `DataFrames` to `GeoDataFrames`. To achieve this, we use `pd.merge` with a `GeoDataFrame` as the first argument and add columns to it from a `DataFrame` specified as the second argument. In the following example, we combine data on coffee production with the `world` dataset. The coffee data is in a `DataFrame` called `coffee_data` imported from a CSV file of major coffee-producing nations.

```
coffee_data = pd.read_csv('data/coffee_data.csv')
coffee_data
```

	name_long	coffee_production_2016	coffee_production_2017
0	Angola	NaN	NaN
1	Bolivia	3.0	4.0
2	Brazil	3277.0	2786.0
...	...	...	...
44	Zambia	3.0	NaN
45	Zimbabwe	1.0	1.0
46	Others	23.0	26.0

Its columns are `name_long`—country name, and `coffee_production_2016` and `coffee_production_2017`—estimated values for coffee production in units of 60-kg bags per year, for 2016 and 2017, respectively.

A left join, which preserves the first dataset, merges `world` with `coffee_data`, based on the common `'name_long'` column:

```
world_coffee = pd.merge(world, coffee_data, on='name_long', how='left')
world_coffee
```

	iso_a2	name_long	...	coffee_production_2016	coffee_production_2017
0	FJ	Fiji	...	NaN	NaN
1	TZ	Tanzania	...	81.0	66.0
2	EH	Western Sahara	...	NaN	NaN
...	...	...	...	...	...
174	XK	Kosovo	...	NaN	NaN
175	TT	Trinidad and Tobago	...	NaN	NaN
176	SS	South Sudan	...	NaN	NaN

The result is a `GeoDataFrame` object identical to the original `world` object, but with two new variables (`coffee_production_2016` and `coffee_production_2017`) on coffee production. This can be plotted as a map, as illustrated (for `coffee_production_2017`) in Figure 2.3. Note that, here and in many other examples in later chapters, we are using a technique to plot two layers (all of the world countries outline, and coffee production with symbology) at once, which will be 'formally' introduced towards the end of the book in Section 8.2.5.

```
base = world_coffee.plot(color='white', edgecolor='lightgrey')
coffee_map = world_coffee.plot(ax=base, column='coffee_production_2017');
```

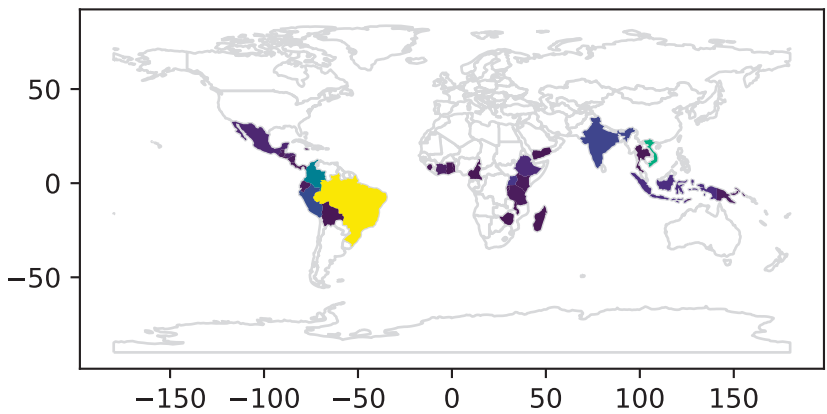


Figure 2.3: World coffee production, thousand 60-kg bags by country, in 2017 (source: International Coffee Organization).

To work, attribute-based joins need a 'key variable' in both datasets (`on` parameter of `pd.merge`). In the above example, both `world_coffee` and `world` DataFrames contained a column called `name_long`.

> **Note**
>
> By default, `pd.merge` uses all columns with matching names. However, it is recommended to explicitly specify the names of the columns to be used for matching, like we did in the last example.

In case where column names are not the same, you can use `left_on` and `right_on` to specify the respective columns.

Note that the result `world_coffee` has the same number of rows as the original dataset `world`. Although there are only 47 rows in `coffee_data`, all 177 country records are kept intact in `world_coffee`. Rows in the original dataset with no match are assigned `np.nan` values for the new coffee production variables. This is a characteristic of a left join (specified with `how='left'`) and is what we typically want to do.

What if we only want to keep countries that have a match in the key variable? In that case an inner join can be used, which keeps only rows with a match in both datasets. We can use it with the `how='inner'` argument.

```
pd.merge(world, coffee_data, on='name_long', how='inner')
```

	iso_a2	name_long	...	coffee_production_2016	coffee_production_2017
0	TZ	Tanzania	...	81.0	66.0
1	PG	Papua New Guinea	...	114.0	74.0
2	ID	Indonesia	...	742.0	360.0
...	...	...	...	...	...
42	ET	Ethiopia	...	215.0	283.0
43	UG	Uganda	...	408.0	443.0
44	RW	Rwanda	...	36.0	42.0

2.2.4 Creating attributes and removing spatial information

Often, we would like to create a new column based on already existing columns. For example, we want to calculate population density for each country. For this we need to divide a population column, here `pop`, by an area column, here `area_km2`. Note that we are working on a copy of `world` named `world2` so that we do not modify the original layer.

```
world2 = world.copy()
world2['pop_dens'] = world2['pop'] / world2['area_km2']
world2
```

	iso_a2	name_long	...	geometry	pop_dens
0	FJ	Fiji	...	MULTIPOLYGON (((-180 -16.55522,...	45.920547
1	TZ	Tanzania	...	MULTIPOLYGON (((33.90371 -0.95,...	56.001184
2	EH	Western Sahara	...	MULTIPOLYGON (((-8.66559 27.656...	NaN
...	...	...	...	...	...
174	XK	Kosovo	...	MULTIPOLYGON (((20.59025 41.855...	162.222400
175	TT	Trinidad and Tobago	...	MULTIPOLYGON (((-61.68 10.76, -...	175.048628
176	SS	South Sudan	...	MULTIPOLYGON (((30.83385 3.5091...	18.452237

To paste (i.e., concatenate) together existing columns, we can use the ordinary Python string operator `+`, as if we are working with individual strings rather than `Series`. For example, we want to combine the `continent` and `region_un` columns into a new column named `con_reg`, using `':'` as a separator. Subsequently, we remove the original columns using `.drop`:

```
world2['con_reg'] = world['continent'] + ':' + world2['region_un']
world2 = world2.drop(['continent', 'region_un'], axis=1)
world2
```

	iso_a2	name_long	...	pop_dens	con_reg
0	FJ	Fiji	...	45.920547	Oceania:Oceania
1	TZ	Tanzania	...	56.001184	Africa:Africa
2	EH	Western Sahara	...	NaN	Africa:Africa
...	...	...	...	...	...
174	XK	Kosovo	...	162.222400	Europe:Europe
175	TT	Trinidad and Tobago	...	175.048628	North America:Americas
176	SS	South Sudan	...	18.452237	Africa:Africa

The resulting `GeoDataFrame` object has a new column called `con_reg` representing the continent and region of each country, e.g., `'South America:Americas'` for Argentina and other South America countries. The opposite operation, splitting one column into multiple columns based on a separator string, is done using the `.str.split` method. As a result, we go back to the previous state of two separate `continent` and `region_un` columns (only that their position is now last, since they are newly created). The `str.split` method returns a column of `lists` by default; to place the strings into separate `str` columns we use the `expand=True` argument.

```
world2[['continent', 'region_un']] = world2['con_reg'] \
    .str.split(':', expand=True)
world2
```

	iso_a2	name_long	...	continent	region_un
0	FJ	Fiji	...	Oceania	Oceania
1	TZ	Tanzania	...	Africa	Africa
2	EH	Western Sahara	...	Africa	Africa
...	...	...	...	...	...
174	XK	Kosovo	...	Europe	Europe
175	TT	Trinidad and Tobago	...	North America	Americas
176	SS	South Sudan	...	Africa	Africa

Renaming one or more columns can be done using the `.rename` method combined with the `columns` argument, which should be a dictionary of the form `old_name:new_name`, as shown above (Section 2.2.1). The following command, for example, renames the lengthy `name_long` column to simply `name`.

```
world2.rename(columns={'name_long': 'name'})
```

	iso_a2	name	...	continent	region_un
0	FJ	Fiji	...	Oceania	Oceania
1	TZ	Tanzania	...	Africa	Africa
2	EH	Western Sahara	...	Africa	Africa
...	...	...	...	...	...
174	XK	Kosovo	...	Europe	Europe
175	TT	Trinidad and Tobago	...	North America	Americas
176	SS	South Sudan	...	Africa	Africa

To change all column names at once, we assign a `list` of the 'new' column names into the `.columns` property. The `list` must be of the same length as the number of columns (i.e., `world.shape[1]`). This is illustrated below, which outputs the same `world2` object, but with very short names.

```
new_names = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'geom', 'i', 'j', 'k', 'l']
world2.columns = new_names
world2
```

	a	b	...	k	l
0	FJ	Fiji	...	Oceania	Oceania
1	TZ	Tanzania	...	Africa	Africa
2	EH	Western Sahara	...	Africa	Africa
...	...	...	...	...	...
174	XK	Kosovo	...	Europe	Europe
175	TT	Trinidad and Tobago	...	North America	Americas
176	SS	South Sudan	...	Africa	Africa

To reorder columns, we can pass a modified columns list to the subsetting operator `[`. For example, the following expressions reorder `world2` columns in reverse alphabetical order.

```
names = sorted(world2.columns, reverse=True)
world2 = world2[names]
world2
```

	l	k	...	b	a
0	Oceania	Oceania	...	Fiji	FJ
1	Africa	Africa	...	Tanzania	TZ
2	Africa	Africa	...	Western Sahara	EH
...	...	...	...	...	...
174	Europe	Europe	...	Kosovo	XK
175	Americas	North America	...	Trinidad and Tobago	TT
176	Africa	Africa	...	South Sudan	SS

Each of these attribute data operations, even though they are defined in the **pandas** package and applicable to any `DataFrame`, preserve the geometry column and the `GeoDataFrame` class. Sometimes, however, it makes sense to remove the geometry, for example to speed-up aggregation or to export just the attribute data for statistical analysis. To go from `GeoDataFrame` to `DataFrame` we need to.

1. Drop the geometry column
2. Convert from `GeoDataFrame` into a `DataFrame`

For example, by the end of the following code section `world2` becomes a regular `DataFrame`.

```
world2 = world2.drop('geom', axis=1)
world2 = pd.DataFrame(world2)
world2
```

	l	k	...	b	a
0	Oceania	Oceania	...	Fiji	FJ
1	Africa	Africa	...	Tanzania	TZ
2	Africa	Africa	...	Western Sahara	EH
...	...	...	...	...	...
174	Europe	Europe	...	Kosovo	XK
175	Americas	North America	...	Trinidad and Tobago	TT
176	Africa	Africa	...	South Sudan	SS

2.3 Manipulating raster objects

Raster cell values can be considered the counterpart of vector attribute values. In this section, we cover operations that deal with raster values in a similar

way, namely as a series of numbers. This type of operations includes subsetting raster values (Section 2.3.1) and calculating global summaries of raster values (Section 2.3.2).

2.3.1 Raster subsetting

When using **rasterio**, raster values are accessible through a **numpy** array, which can be imported with the `.read` method (as we saw in Section 1.3.1). As shown in Section 1.3.1, reading a single raster layer (or the only layer of a single-band raster, such as here) results in a two-dimensional array:

```
elev = src_elev.read(1)
elev
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35, 36]], dtype=uint8)
```

Then, we can access any subset of cell values using **numpy** methods, keeping in mind that dimensions order is `(rows, columns)`. For example, `elev[1,2]` returns the value at row 2, column 3.

```
elev[1, 2]
```

```
np.uint8(9)
```

Cell values can be modified by overwriting existing values in conjunction with a subsetting operation, e.g., `elev[1,2]=0` to set cell at row 2, column 3 of `elev` to 0.

```
elev[1, 2] = 0
elev
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  0, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35, 36]], dtype=uint8)
```

Multiple cells can also be modified in this way, e.g., `elev[0,0:3]=0` to set the first three cells in the first row to 0.

```
elev[0, 0:3] = 0
elev
```

```
array([[ 0,  0,  0,  4,  5,  6],
       [ 7,  8,  0, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35, 36]], dtype=uint8)
```

Alternatively, reading more than one layer, or all layers (even if there is just one, such as here) results in a three-dimensional array.

```
elev3d = src_elev.read()
elev3d
```

```
array([[[ 1,  2,  3,  4,  5,  6],
        [ 7,  8,  9, 10, 11, 12],
        [13, 14, 15, 16, 17, 18],
        [19, 20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29, 30],
        [31, 32, 33, 34, 35, 36]]], dtype=uint8)
```

> **Note**
>
> You can see that the above array is three-dimensional according to the number of brackets `[`, or check explicitly using `.shape` or `.ndim`.

In three-dimensional arrays, we access cell values using three indices, keeping in mind that dimensions order is `(layers, rows, columns)` For example, to get the same value shown above, at row 2, column 3 (at band 1), we use `elev[0,1,2]` instead of `elev[1,2]`.

```
elev3d[0, 1, 2]
```

```
np.uint8(9)
```

2.3.2 Summarizing raster objects

Global summaries of raster values can be calculated by applying **numpy** summary functions on the array with raster values, e.g., `np.mean`.

```
np.mean(elev)
```

```
np.float64(18.083333333333332)
```

Note that 'No Data'-safe functions–such as `np.nanmean`—should be used in case the raster contains 'No Data' values which need to be ignored. Before we can demonstrate that, we must convert the array from `int` to `float`, as `int` arrays cannot contain `np.nan` (due to computer memory limitations).

```
elev1 = elev.copy()
elev1 = elev1.astype('float64')
elev1
```

```
array([[ 0.,  0.,  0.,  4.,  5.,  6.],
       [ 7.,  8.,  0., 10., 11., 12.],
       [13., 14., 15., 16., 17., 18.],
       [19., 20., 21., 22., 23., 24.],
       [25., 26., 27., 28., 29., 30.],
       [31., 32., 33., 34., 35., 36.]])
```

Now we can insert an `np.nan` value into the array, for example to a cell located in the first row and third column. (Doing so in the original `elev` array raises an error, because an `int` array cannot accommodate `np.nan`, as mentioned above; try it to see for yourself.)

```
elev1[0, 2] = np.nan
elev1
```

```
array([[ 0.,  0., nan,  4.,  5.,  6.],
       [ 7.,  8.,  0., 10., 11., 12.],
       [13., 14., 15., 16., 17., 18.],
       [19., 20., 21., 22., 23., 24.],
       [25., 26., 27., 28., 29., 30.],
       [31., 32., 33., 34., 35., 36.]])
```

With the `np.nan` value in place, the `np.mean` summary value becomes unknown (`np.nan`).

```
np.mean(elev1)
```

```
np.float64(nan)
```

To get a summary of all non-missing values, we need to use one of the specialized **numpy** functions that ignore 'No Data' values, such as `np.nanmean`:

```
np.nanmean(elev1)
```

```
np.float64(18.6)
```

Raster value statistics can be visualized in a variety of ways. One approach is to 'flatten' the raster values into a one-dimensional array (using `.flatten`), then use a graphical function such as `plt.hist` or `plt.boxplot` (from **matplotlib.pyplot**). For example, the following code section shows the distribution of values in `elev` using a histogram (Figure 2.4).

```
plt.hist(elev.flatten());
```

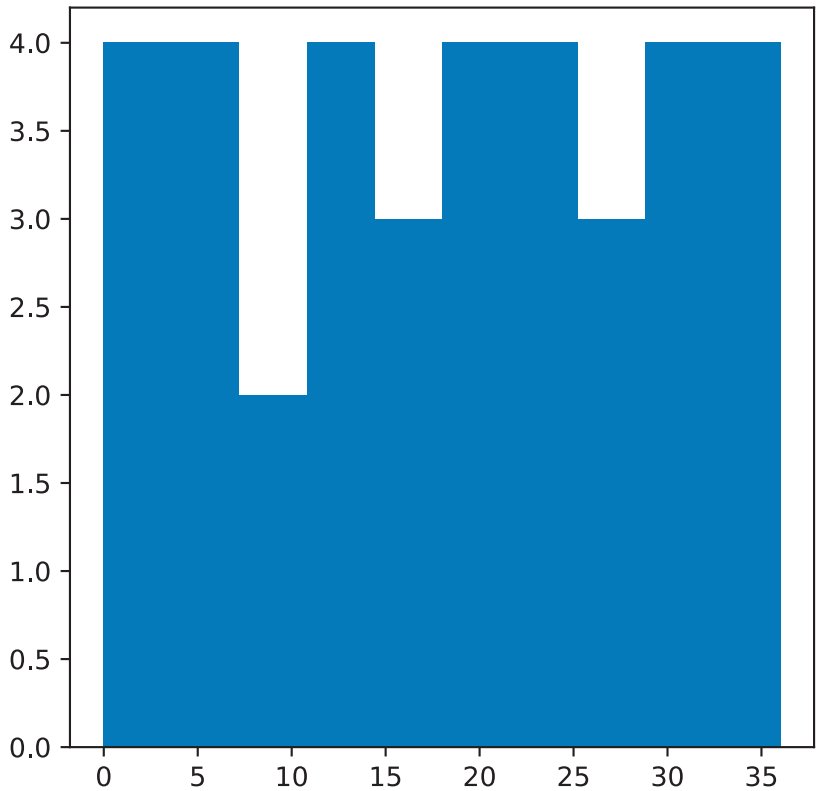


Figure 2.4: Distribution of cell values in a continuous raster (`elev.tif`)

To summarize the distribution of values in a categorical raster, we can calculate the frequencies of unique values and draw them using a barplot. Let's demonstrate using the `grain.tif` small categorical raster.

```
grain = src_grain.read(1)
grain
```

```
array([[1, 0, 1, 2, 2, 2],
       [0, 2, 0, 0, 2, 1],
       [0, 2, 2, 0, 0, 2],
       [0, 0, 1, 1, 1, 1],
       [1, 1, 1, 2, 1, 1],
       [2, 1, 2, 2, 0, 2]], dtype=uint8)
```

To calculate the frequency of unique values in an array, we use the `np.unique` with the `return_counts=True` option. The result is a `tuple` with two corresponding arrays: the unique values, and their counts.

```
freq = np.unique(grain, return_counts=True)
freq
```

```
(array([0, 1, 2], dtype=uint8), array([10, 13, 13]))
```

These two arrays can be passed to the `plt.bar` function to draw a barplot, as shown in Figure 2.5.

```
plt.bar(*freq);
```

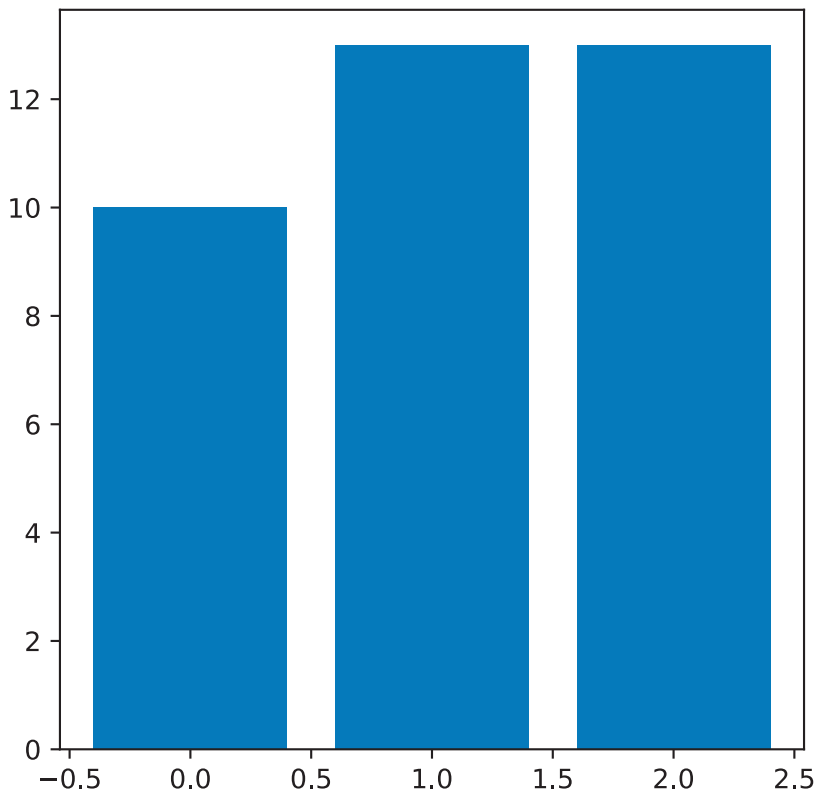


Figure 2.5: Distribution of cell values in categorical raster (`grain.tif`)

3

Spatial data operations

Prerequisites

This chapter requires importing the following packages:

```
import os
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy.ndimage
import scipy.stats
import shapely
import geopandas as gpd
import rasterio
import rasterio.plot
import rasterio.merge
import rasterio.features
```

It also relies on the following data files:

```
nz = gpd.read_file('data/nz.gpkg')
nz_height = gpd.read_file('data/nz_height.gpkg')
world = gpd.read_file('data/world.gpkg')
cycle_hire = gpd.read_file('data/cycle_hire.gpkg')
cycle_hire_osm = gpd.read_file('data/cycle_hire_osm.gpkg')
src_elev = rasterio.open('output/elev.tif')
src_landsat = rasterio.open('data/landsat.tif')
src_grain = rasterio.open('output/grain.tif')
```

3.1 Introduction

Spatial operations, including spatial joins between vector datasets and local and focal operations on raster datasets, are a vital part of geocomputation.

DOI: 10.1201/9781003379911-3

This chapter shows how spatial objects can be modified in a multitude of ways based on their location and shape. Many spatial operations have a non-spatial (attribute) equivalent, so concepts such as subsetting and joining datasets demonstrated in the previous chapter are applicable here. This is especially true for vector operations: Section 2.2 on vector attribute manipulation provides the basis for understanding its spatial counterpart, namely spatial subsetting (covered in Section 3.2.1). Spatial joining (Section 3.2.3) and aggregation (Section 3.2.5) also have non-spatial counterparts, covered in the previous chapter.

Spatial operations differ from non-spatial operations in a number of ways, however. Spatial joins, for example, can be done in a number of ways—including matching entities that intersect with or are within a certain distance of the target dataset—while the attribution joins discussed in Section 2.2.3 in the previous chapter can only be done in one way. Different types of spatial relationships between objects, including intersects and disjoints, are described in Section 3.2.2. Another unique aspect of spatial objects is distance: all spatial objects are related through space, and distance calculations can be used to explore the strength of this relationship, as described in the context of vector data in Section 3.2.7.

Spatial operations on raster objects include subsetting—covered in Section 3.3.1—and merging several raster 'tiles' into a single object, as demonstrated in Section 3.3.8. Map algebra covers a range of operations that modify raster cell values, with or without reference to surrounding cell values. The concept of map algebra, vital for many applications, is introduced in Section 3.3.2; local, focal, and zonal map algebra operations are covered in Section 3.3.3, Section 3.3.4, and Section 3.3.5, respectively. Global map algebra operations, which generate summary statistics representing an entire raster dataset, and distance calculations on rasters, are discussed in Section 3.3.6.

> **Note**
>
> It is important to note that spatial operations that use two spatial objects rely on both objects having the same coordinate reference system, a topic that was introduced in Section 1.4 and which will be covered in more depth in Chapter 6.

3.2 Spatial operations on vector data

This section provides an overview of spatial operations on vector geographic data represented as Simple Features using the **shapely** and **geopandas**

packages. Section 3.3 then presents spatial operations on raster datasets, using the **rasterio** and **scipy** packages.

3.2.1 Spatial subsetting

Spatial subsetting is the process of taking a spatial object and returning a new object containing only features that relate in space to another object. Analogous to attribute subsetting (covered in Section 2.2.1), subsets of `GeoDataFrame`s can be created with square bracket (`[]`) operator using the syntax `x[y]`, where `x` is an `GeoDataFrame` from which a subset of rows/features will be returned, and `y` is a boolean `Series`. The difference is, that, in spatial subsetting `y` is created based on another geometry and using one of the binary geometry relation methods, such as `.intersects` (see Section 3.2.2), rather than based on comparison based on ordinary columns.

To demonstrate spatial subsetting, we will use the `nz` and `nz_height` layers, which contain geographic data on the 16 main regions and 101 highest points in New Zealand, respectively (Figure 3.1 (a)), in a projected coordinate system. The following expression creates a new object, `canterbury`, representing only one region—Canterbury.

```
canterbury = nz[nz['Name'] == 'Canterbury']
canterbury
```

	Name	Island	...	Sex_ratio	geometry
10	Canterbury	South	...	0.975327	MULTIPOLYGON (((1686901.914 535...

Then, we use the `.intersects` method to evaluate, for each of the `nz_height` points, whether they intersect with Canterbury. The result `canterbury_height` is a boolean `Series` with the 'answers'.

```
sel = nz_height.intersects(canterbury.geometry.iloc[0])
sel
```

```
0      False
1      False
2      False
       ...
98     False
99     False
100    False
Length: 101, dtype: bool
```

Finally, we can subset `nz_height` using the obtained `Series`, resulting in the subset `canterbury_height` with only those points that intersect with Canterbury.

```
canterbury_height = nz_height[sel]
canterbury_height
```

	t50_fid	elevation	geometry
4	2362630	2749	POINT (1378169.6 5158491.453)
5	2362814	2822	POINT (1389460.041 5168749.086)
6	2362817	2778	POINT (1390166.225 5169466.158)
...	...	...	...
92	2380298	2877	POINT (1652788.127 5348984.469)
93	2380300	2711	POINT (1654213.379 5349962.973)
94	2380308	2885	POINT (1654898.622 5350462.779)

Figure 3.1 compares the original `nz_height` layer (left) with the subset `canterbury_height` (right).

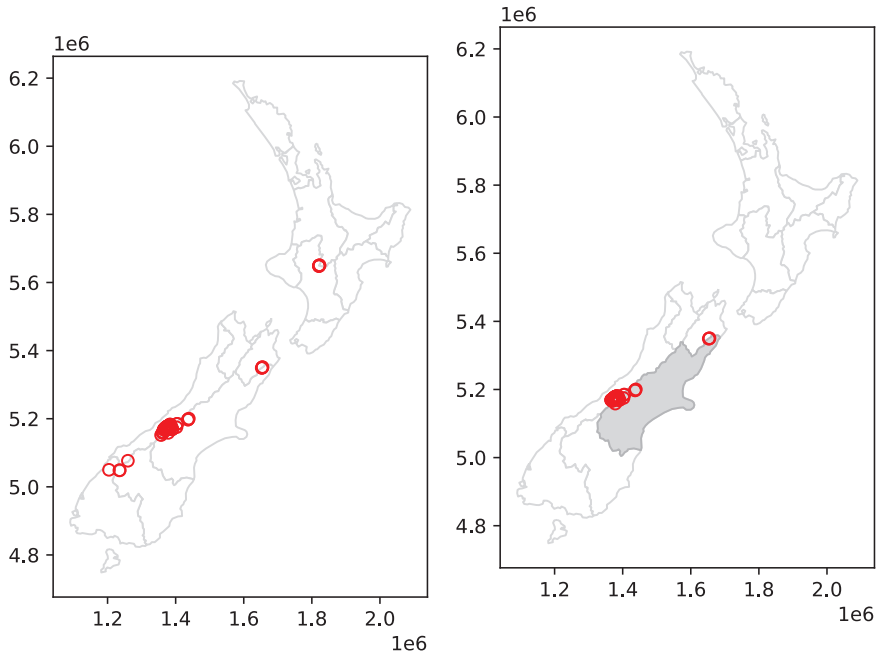
```
# Original
base = nz.plot(color='white', edgecolor='lightgrey')
nz_height.plot(ax=base, color='None', edgecolor='red');
# Subset (intersects)
base = nz.plot(color='white', edgecolor='lightgrey')
canterbury.plot(ax=base, color='lightgrey', edgecolor='darkgrey')
canterbury_height.plot(ax=base, color='None', edgecolor='red');
```

Like in attribute subsetting (Section 2.2.1), we are using a boolean series (`sel`), of the same length as the number of rows in the filtered table (`nz_height`), created based on a condition applied on itself. The difference is that the condition is not a comparison of attribute values, but an evaluation of a spatial relation. Namely, we evaluate whether each geometry of `nz_height` intersects with the `canterbury` geometry, using the `.intersects` method.

Various topological relations can be used for spatial subsetting which determine the type of spatial relationship that features in the target object must have with the subsetting object to be selected. These include touches, crosses, or within, as we will see shortly in Section 3.2.2. Intersects (`.intersects`), which we used in the last example, is the most commonly used method. This is a 'catch all' topological relation, that will return features in the target that touch, cross or are within the source 'subsetting' object. As an example of another method, we can use `.disjoint` to obtain all points that *do not* intersect with Canterbury.

```
sel = nz_height.disjoint(canterbury.geometry.iloc[0])
canterbury_height2 = nz_height[sel]
```

The results are shown in Figure 3.2, which compares the original `nz_height` layer (left) with the subset `canterbury_height2` (right).



(a) Original points (red)

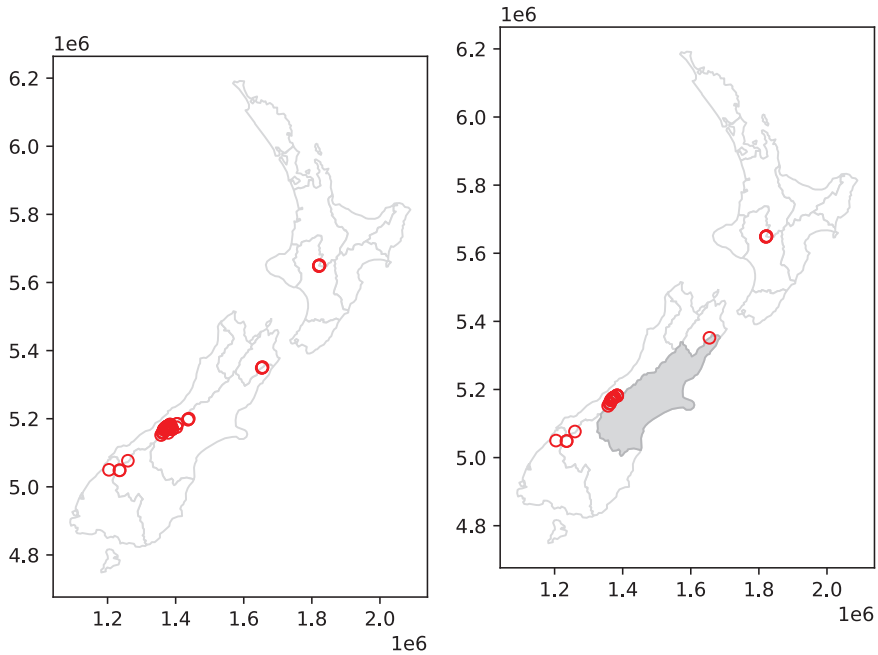
(b) Spatial subset based on intersection (red), geometry used for subsetting (Canterbury) (grey)

Figure 3.1: Spatial subsetting of points by intersection with polygon

```
# Original
base = nz.plot(color='white', edgecolor='lightgrey')
nz_height.plot(ax=base, color='None', edgecolor='red');
# Subset (disjoint)
base = nz.plot(color='white', edgecolor='lightgrey')
canterbury.plot(ax=base, color='lightgrey', edgecolor='darkgrey')
canterbury_height2.plot(ax=base, color='None', edgecolor='red');
```

In case we need to subset according to several geometries at once, e.g., find out which points intersect with both Canterbury and Southland, we can dissolve the filtering subset, using `.union_all`, before applying the `.intersects` (or any other) operator. For example, here is how we can subset the `nz_height` points which intersect with Canterbury or Southland. (Note that we are also using the `.isin` method, as demonstrated at the end of Section 2.2.1.)

```
canterbury_southland = nz[nz['Name'].isin(['Canterbury', 'Southland'])]
sel = nz_height.intersects(canterbury_southland.union_all())
canterbury_southland_height = nz_height[sel]
canterbury_southland_height
```



(a) Original points (red)

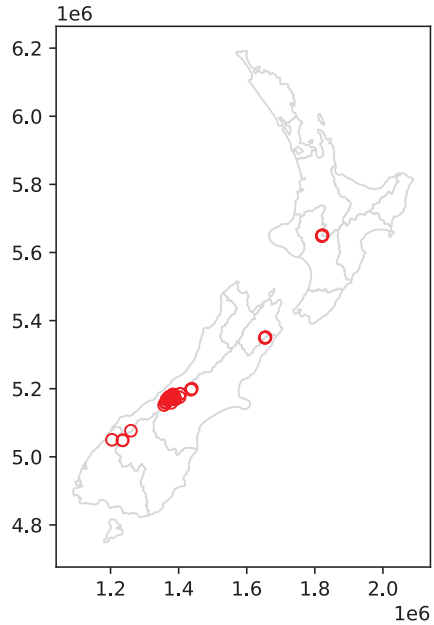
(b) Spatial subset based on disjoint (red), geometry used for subsetting (Canterbury) (grey)

Figure 3.2: Spatial subsetting of points disjoint from a polygon

	t50_fid	elevation	geometry
0	2353944	2723	POINT (1204142.603 5049971.287)
4	2362630	2749	POINT (1378169.6 5158491.453)
5	2362814	2822	POINT (1389460.041 5168749.086)
...	...	...	...
92	2380298	2877	POINT (1652788.127 5348984.469)
93	2380300	2711	POINT (1654213.379 5349962.973)
94	2380308	2885	POINT (1654898.622 5350462.779)

Figure 3.3 shows the results of the spatial subsetting of `nz_height` points by intersection with Canterbury and Southland.

```
# Original
base = nz.plot(color='white', edgecolor='lightgrey')
nz_height.plot(ax=base, color='None', edgecolor='red');
# Subset by intersection with two polygons
base = nz.plot(color='white', edgecolor='lightgrey')
canterbury_southland.plot(ax=base, color='lightgrey', edgecolor='darkgrey')
canterbury_southland_height.plot(ax=base, color='None', edgecolor='red');
```



(a) Original points (red)

(b) Spatial subset based on intersection (red), geometry used for subsetting (Canterbury and Southland) (grey)

Figure 3.3: Spatial subsetting of points by intersection with more than one polygon

The next section further explores different types of spatial relations, also known as binary predicates (of which `.intersects` and `.disjoint` are two examples), that can be used to identify whether or not two features are spatially related.

3.2.2 Topological relations

Topological relations describe the spatial relationships between objects. 'Binary topological relationships', to give them their full name, are logical statements (in that the answer can only be `True` or `False`) about the spatial relationships between two objects defined by ordered sets of points (typically forming points, lines, and polygons) in two or more dimensions (Egenhofer and Herring 1990). That may sound rather abstract and, indeed, the definition and classification of topological relations is based on mathematical foundations first published in book form in 1966 (Spanier 1995), with the field of algebraic topology continuing into the 21st century (Dieck 2008).

Despite their mathematical origins, topological relations can be understood intuitively with reference to visualizations of commonly used functions that

test for common types of spatial relationships. Figure 3.4 shows a variety of geometry pairs and their associated relations. The third and fourth pairs in Figure 3.4 (from left to right and then down) demonstrate that, for some relations, order is important: while the relations equals, intersects, crosses, touches and overlaps are symmetrical, meaning that if `x.relation(y)` is true, `y.relation(x)` will also be true, relations in which the order of the geometries are important such as contains and within are not.

> **Note**
>
> Notice that each geometry pair has a 'DE-9IM'[1] string such as `FF2F11212`. DE-9IM strings describe the dimensionality (0=points, 1=lines, 2=polygons) of the pairwise intersections of the interior, boundary, and exterior, of two geometries (i.e., nine values of 0/1/2 encoded into a string). This is an advanced topic beyond the scope of this book, which can be useful to understand the difference between relation types, or define custom types of relations. See the DE-9IM strings section in Geocomputation with R (Lovelace, Nowosad, and Muenchow 2019). Also note that the **shapely** package contains the `.relate` and `.relate_pattern` methods, to derive and to test for DE-9IM patterns, respectively.

In **shapely**, methods testing for different types of topological relations are known as 'relationships'. **geopandas** provides their wrappers (with the same method name) which can be applied on multiple geometries at once (such as `.intersects` and `.disjoint` applied on all points in `nz_height`, see Section 3.2.1). To see how topological relations work in practice, let's create a simple reproducible example, building on the relations illustrated in Figure 3.4 and consolidating knowledge of how vector geometries are represented from a previous chapter (Section 1.2.3 and Section 1.2.5).

```
points = gpd.GeoSeries([
  shapely.Point(0.2,0.1),
  shapely.Point(0.7,0.2),
  shapely.Point(0.4,0.8)
])
line = gpd.GeoSeries([
  shapely.LineString([(0.4,0.2), (1,0.5)])
])
poly = gpd.GeoSeries([
  shapely.Polygon([(0,0), (0,1), (1,1), (1,0.5), (0,0)])
])
```

[1] https://en.wikipedia.org/wiki/DE-9IM

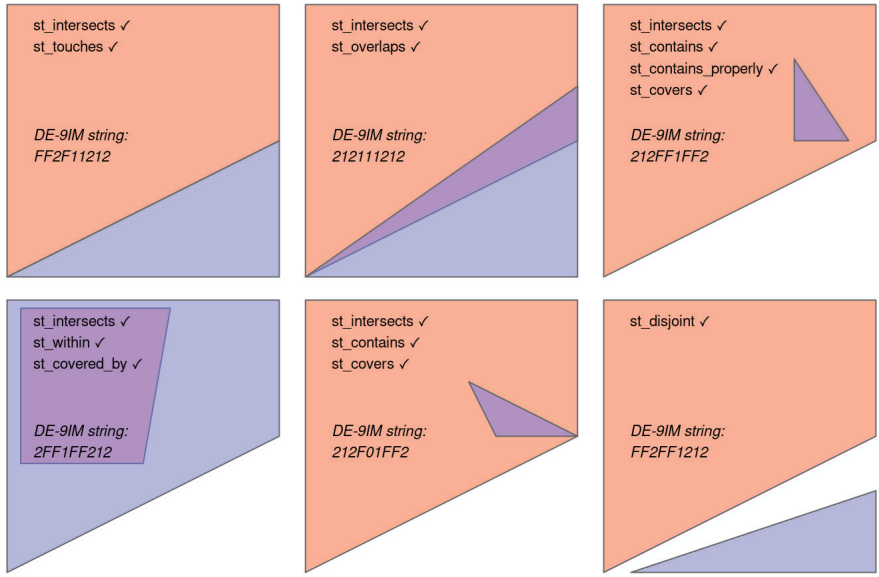


Figure 3.4: Topological relations between vector geometries, inspired by Figures 1 and 2 in Egenhofer and Herring (1990). The relations for which the `x.relation(y)` is true are printed for each geometry pair, with `x` represented in pink and `y` represented in blue. The nature of the spatial relationship for each pair is described by the Dimensionally Extended 9-Intersection Model string.

The sample dataset which we created is composed of three `GeoSeries`: named `points`, `line`, and `poly`, which are visualized in Figure 3.5. The last expression is a `for` loop used to add text labels (`0`, `1`, and `2`) to identify the points; we are going to explain the concepts of text annotations with **geopandas** `.plot` in Section 8.2.4.

```
base = poly.plot(color='lightgrey', edgecolor='red')
line.plot(ax=base, color='black', linewidth=7)
points.plot(ax=base, color='none', edgecolor='black')
for i in enumerate(points):
    base.annotate(
        i[0], xy=(i[1].x, i[1].y),
        xytext=(3, 3), textcoords='offset points', weight='bold'
    )
```

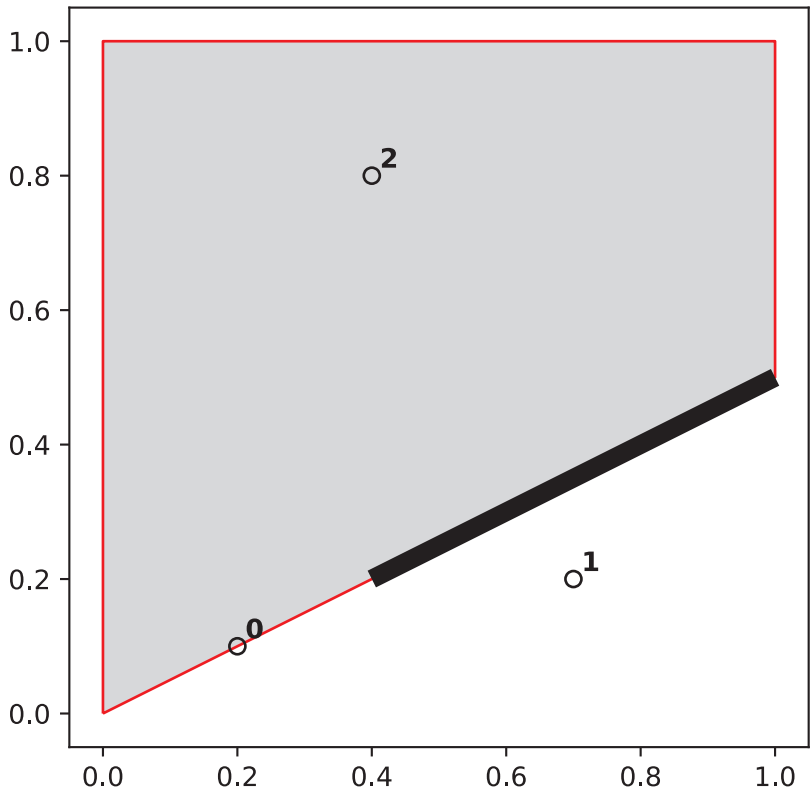


Figure 3.5: Points (`points`), line (`line`), and polygon (`poly`) objects used to illustrate topological relations

A simple query is: which of the points in `points` intersect in some way with polygon `poly`? The question can be answered by visual inspection (points `0` and `2` are touching and are within the polygon, respectively). Alternatively, we can get the solution with the `.intersects` method, which reports whether or not each geometry in a `GeoSeries` (`points`) intersects with a single `shapely` geometry (`poly.iloc[0]`).

```
points.intersects(poly.iloc[0])
```

```
0     True
1    False
2     True
dtype: bool
```

The result shown above is a boolean `Series`. Its contents should match our intuition: positive (`True`) results are returned for the points `0` and `2`, and a

negative result (`False`) for point `1`. Each value in this `Series` represents a feature in the first input (`points`).

All earlier examples in this chapter demonstrate the 'many-to-one' mode of `.intersects` and analogous methods, where the relation is evaluated between each of several geometries in a `GeoSeries`/`GeoDataFrame`, and an individual `shapely` geometry. A second mode of those methods (not demonstrated here) is when both inputs are `GeoSeries`/`GeoDataFrame` objects. In such case, a 'pairwise' evaluation takes place between geometries aligned by index (`align=True`, the default) or by position (`align=False`). For example, the expression `nz.intersects(nz)` returns a `Series` of 16 `True` values, indicating (unsurprisingly) that each geometry in `nz` intersects with itself.

A third mode is when we are interested in a 'many-to-many' evaluation, i.e., obtaining a matrix of all pairwise combinations of geometries from two `GeoSeries` objects. At the time of writing, there is no built-in method to do this in **geopandas**. However, the `.apply` method (package **pandas**) can be used to repeat a 'many-to-one' evaluation over all geometries in the second layer, resulting in a matrix of *pairwise* results. We will create another `GeoSeries` with two polygons, named `poly2`, to demonstrate this.

```
poly2 = gpd.GeoSeries([
  shapely.Polygon([(0,0), (0,1), (1,1), (1,0.5), (0,0)]),
  shapely.Polygon([(0,0), (1,0.5), (1,0), (0,0)])
])
```

Our two input objects, `points` and `poly2`, are illustrated in Figure 3.6.

```
base = poly2.plot(color='lightgrey', edgecolor='red')
points.plot(ax=base, color='none', edgecolor='black')
for i in enumerate(points):
    base.annotate(
        i[0], xy=(i[1].x, i[1].y),
        xytext=(3, 3), textcoords='offset points', weight='bold'
    )
```

Now we can use `.apply` to get the intersection relations matrix. The result is a `DataFrame`, where each row represents a `points` geometry and each column represents a `poly2` geometry. We can see that the point `0` intersects with both polygons, while points `1` and `2` intersect with one of the polygons each.

```
points.apply(lambda x: poly2.intersects(x))
```

	0	1
0	True	True
1	False	True
2	True	False

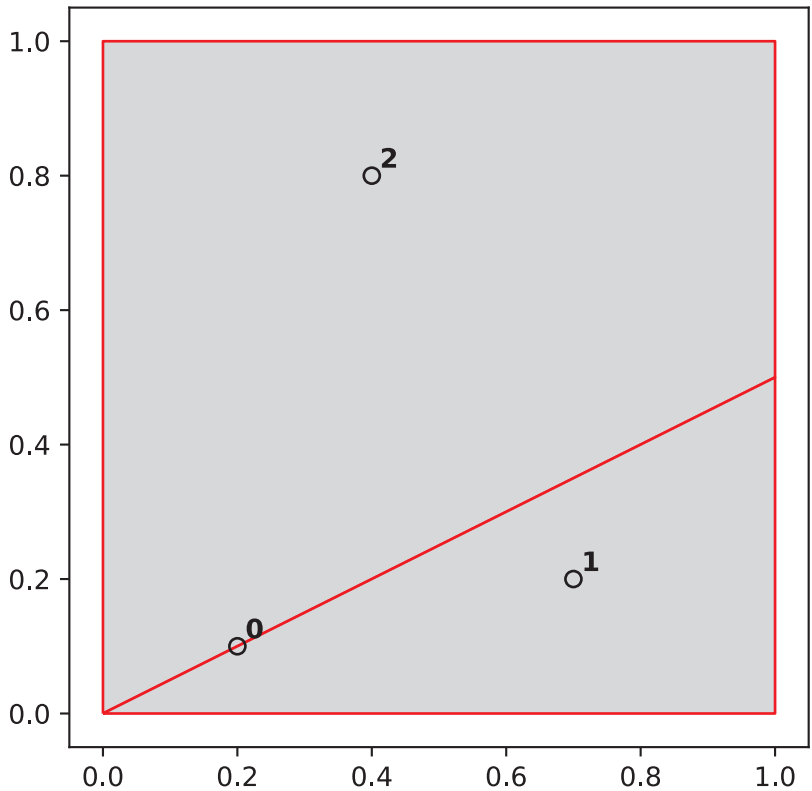


Figure 3.6: Inputs for demonstrating the evaluation of all pairwise intersection relations between three points (`points`) and two polygons (`poly2`)

> **Note**
>
> The `.apply` method (package **pandas**) is used to apply a function along one of the axes of a `DataFrame` (or `GeoDataFrame`). That is, we can apply a function on all rows (`axis=1`) or all columns (`axis=0`, the default). When the function being applied returns a single value, the output of `.apply` is a `Series` (e.g., `.apply(len)` returns the lengths of all columns, because `len` returns a single value). When the function returns a `Series`, then `.apply` returns a `DataFrame` (such as in the above example.)

> **Note**
>
> Since the above result, like any pairwise matrix, (1) is composed of values of the same type, and (2) has no contrasting role for rows and columns, is may be more convenient to use a plain **numpy** array to work with it. In such case, we can use the `.to_numpy` method to go from `DataFrame` to `ndarray`.

```
points.apply(lambda x: poly2.intersects(x)).to_numpy()
```

```
array([[ True,  True],
       [False,  True],
       [ True, False]])
```

The `.intersects` method returns `True` even in cases where the features just touch: intersects is a 'catch-all' topological operation which identifies many types of spatial relations, as illustrated in Figure 3.4. More restrictive questions include which points lie within the polygon, and which features are on or contain a shared boundary with it? The first question can be answered with `.within`, and the second with `.touches`.

```
points.within(poly.iloc[0])
```

```
0    False
1    False
2     True
dtype: bool
```

```
points.touches(poly.iloc[0])
```

```
0     True
1    False
2    False
dtype: bool
```

Note that although the point `0` touches the boundary polygon, it is not within it; point `2` is within the polygon but does not touch any part of its border. The opposite of `.intersects` is `.disjoint`, which returns only objects that do not spatially relate in any way to the selecting object.

```
points.disjoint(poly.iloc[0])
```

```
0    False
1     True
2    False
dtype: bool
```

Another useful type of relation is 'within distance', where we detect features that intersect with the target buffered by particular distance. Buffer distance determines how close target objects need to be before they are selected. This can be done by literally buffering (Section 1.2.5) the target geometry, and evaluating intersection (`.intersects`). Another way is to calculate the distances using the `.distance` method, and then evaluate whether they are within a threshold distance.

```
points.distance(poly.iloc[0]) < 0.2
```

```
0    True
1    True
2    True
dtype: bool
```

Note that although point `1` is more than `0.2` units of distance from the nearest vertex of `poly`, it is still selected when the distance is set to `0.2`. This is because distance is measured to the nearest edge, in this case, the part of the polygon that lies directly above point 2 in Figure 3.4. We can verify that the actual distance between point `1` and the polygon is `0.13`, as follows.

```
points.iloc[1].distance(poly.iloc[0])
```

```
0.13416407864998736
```

This is also a good opportunity to repeat that all distance-related calculations in **geopandas** (and **shapely**) assume planar geometry, and only take into account the coordinate values. It is up to the user to make sure that all input layers are in the same projected CRS, so that this type of calculations make sense (see Section 6.4 and Section 6.5).

3.2.3 Spatial joining

Joining two non-spatial datasets uses a shared 'key' variable, as described in Section 2.2.3. Spatial data joining applies the same concept, but instead relies on spatial relations, described in the previous section. As with attribute data, joining adds new columns to the target object (the argument `x` in joining functions), from a source object (`y`).

The following example illustrates the process: imagine you have ten points randomly distributed across the Earth's surface and you ask, for the points that are on land, which countries are they in? Implementing this idea in a reproducible example will build your geographic data handling skills and show how spatial joins work. The starting point is to create points that are randomly scattered over the planar surface that represents Earth's geographic coordinates, in decimal degrees (Figure 3.7 (a)).

```
np.random.seed(11)       ## set seed for reproducibility
bb = world.total_bounds  ## the world's bounds
x = np.random.uniform(low=bb[0], high=bb[2], size=10)
y = np.random.uniform(low=bb[1], high=bb[3], size=10)
random_points = gpd.points_from_xy(x, y, crs=4326)
random_points = gpd.GeoDataFrame({'geometry': random_points})
random_points
```

	geometry
0	POINT (-115.10291 36.78178)
1	POINT (-172.98891 -71.02938)
2	POINT (-13.24134 65.23272)
...	...
7	POINT (-4.54623 -69.64082)
8	POINT (159.05039 -34.99599)
9	POINT (126.28622 -62.49509)

The scenario illustrated in Figure 3.7 shows that the `random_points` object (top left) lacks attribute data, while the world (top right) has attributes, including country names that are shown for a sample of countries in the legend. Before creating the joined dataset, we use spatial subsetting to create `world_random`, which contains only countries that contain random points, to verify the number of country names returned in the joined dataset should be four (see Figure 3.7 (b)).

```
world_random = world[world.intersects(random_points.union_all())]
world_random
```

	iso_a2	name_long	...	gdpPercap	geometry
4	US	United States	...	51921.984639	MULTIPOLYGON (((-171.73166 63.7...
18	RU	Russian Federation	...	25284.586202	MULTIPOLYGON (((-180 64.97971, ...
52	ML	Mali	...	1865.160622	MULTIPOLYGON (((-11.51394 12.44...
159	AQ	Antarctica	...	NaN	MULTIPOLYGON (((-180 -89.9, 179...

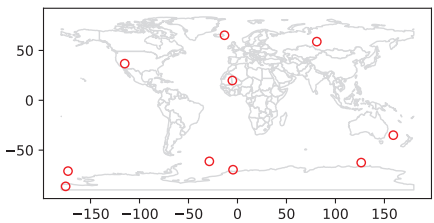
Spatial joins are implemented with `x.sjoin(y)`, as illustrated in the code chunk below. The output is the `random_joined` object which is illustrated in Figure 3.7 (c).

```
random_joined = random_points.sjoin(world, how='left')
random_joined
```

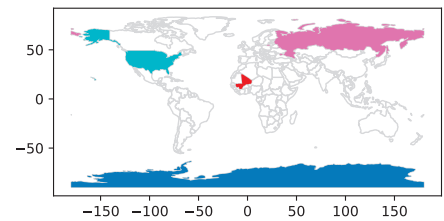
	geometry	index_right	...	lifeExp	gdpPercap
0	POINT (-115.10291 36.78178)	4.0	...	78.841463	51921.984639
1	POINT (-172.98891 -71.02938)	NaN	...	NaN	NaN
2	POINT (-13.24134 65.23272)	NaN	...	NaN	NaN
...	...	...	...	...	...
7	POINT (-4.54623 -69.64082)	NaN	...	NaN	NaN
8	POINT (159.05039 -34.99599)	NaN	...	NaN	NaN
9	POINT (126.28622 -62.49509)	NaN	...	NaN	NaN

Figure 3.7 shows the input points and countries, the illustration of intersecting countries, and the join result.

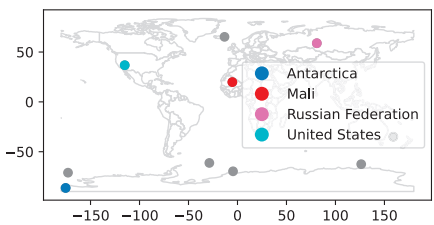
```
# Random points
base = world.plot(color='white', edgecolor='lightgrey')
random_points.plot(ax=base, color='None', edgecolor='red');
# World countries intersecting with the points
base = world.plot(color='white', edgecolor='lightgrey')
world_random.plot(ax=base, column='name_long');
# Points with joined country names
base = world.plot(color='white', edgecolor='lightgrey')
random_joined.geometry.plot(ax=base, color='grey')
random_joined.plot(ax=base, column='name_long', legend=True);
```



(a) A new attribute variable is added to random points,



(b) from source world object,



(c) resulting in points associated with country names

Figure 3.7: Illustration of a spatial join

3.2.4 Non-overlapping joins

Sometimes two geographic datasets do not touch but still have a strong geographic relationship. The datasets `cycle_hire` and `cycle_hire_osm` provide a good example. Plotting them reveals that they are often closely related but they do not seem to touch, as shown in Figure 3.8.

```
base = cycle_hire.plot(edgecolor='blue', color='none')
cycle_hire_osm.plot(ax=base, edgecolor='red', color='none');
```

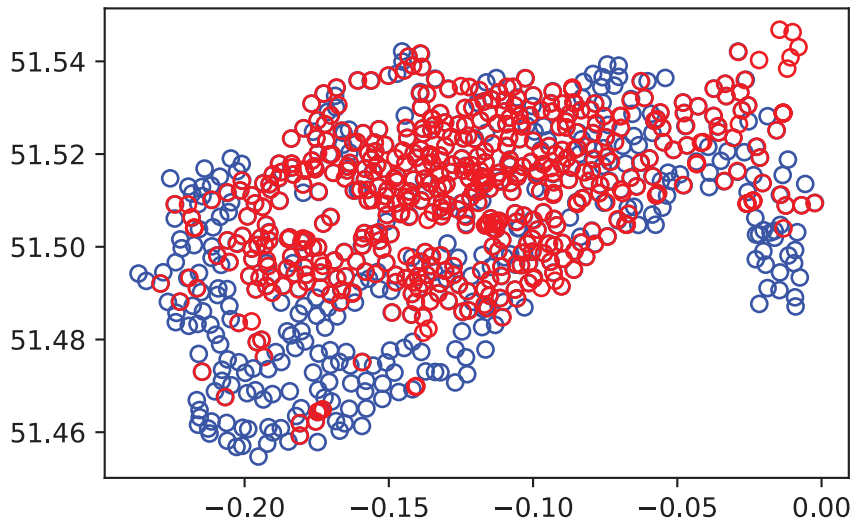


Figure 3.8: The spatial distribution of cycle hire points in London based on official data (blue) and OpenStreetMap data (red).

We can check if any of the points are the same by creating a pairwise boolean matrix of `.intersects` relations, then evaluating whether any of the values in it is `True`. Note that the `.to_numpy` method is applied to go from a `DataFrame` to an `ndarray`, for which `.any` gives a global rather than a row-wise summary. This is what we want in this case, because we are interested in whether any of the points intersect, not whether any of the points in each row intersect.

```
m = cycle_hire.geometry.apply(
  lambda x: cycle_hire_osm.geometry.intersects(x)
)
m.to_numpy().any()
```

```
np.False_
```

Imagine that we need to join the capacity variable in `cycle_hire_osm` (`'capacity'`) onto the official 'target' data contained in `cycle_hire`, which looks as follows.

```
cycle_hire
```

	id	name	...	nempty	geometry
0	1	River Street	...	14	POINT (-0.10997 51.52916)
1	2	Phillimore Gardens	...	34	POINT (-0.19757 51.49961)
2	3	Christopher Street	...	32	POINT (-0.08461 51.52128)
...	...	...	...	...	...
739	775	Little Brook Green	...	17	POINT (-0.22387 51.49666)
740	776	Abyssinia Close	...	10	POINT (-0.16703 51.46033)
741	777	Limburg Road	...	11	POINT (-0.1653 51.46192)

This is when a non-overlapping join is needed. Spatial join (`gpd.sjoin`) along with buffered geometries (see Section 4.2.3) can be used to do that, as demonstrated below using a threshold distance of 20 m. Note that we transform the data to a projected CRS (`27700`) to use real buffer distances, in meters (see Section 6.4).

```
crs = 27700
cycle_hire_buffers = cycle_hire.copy().to_crs(crs)
cycle_hire_buffers.geometry = cycle_hire_buffers.buffer(20)
cycle_hire_buffers = gpd.sjoin(
    cycle_hire_buffers,
    cycle_hire_osm.to_crs(crs),
    how='left'
)
cycle_hire_buffers
```

	id	name_left	...	cyclestreets_id	description
0	1	River Street	...	None	None
1	2	Phillimore Gardens	...	None	None
2	3	Christopher Street	...	None	None
...	...	...	...	...	...
739	775	Little Brook Green	...	NaN	NaN
740	776	Abyssinia Close	...	NaN	NaN
741	777	Limburg Road	...	NaN	NaN

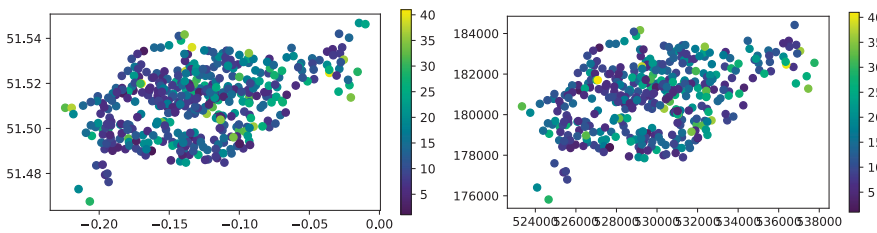
Note that the number of rows in the joined result is greater than the target. This is because some cycle hire stations in `cycle_hire_buffers` have multiple matches in `cycle_hire_osm`. To aggregate the values for the overlapping points and return the mean, we can use the aggregation methods shown in Section 2.2.2, resulting in an object with the same number of rows as the target. We also go back from buffers to points using `.centroid` method.

```
cycle_hire_buffers = cycle_hire_buffers[['id', 'capacity', 'geometry']] \
    .dissolve(by='id', aggfunc='mean') \
    .reset_index()
cycle_hire_buffers.geometry = cycle_hire_buffers.centroid
cycle_hire_buffers
```

	id	geometry	capacity
0	1	POINT (531203.517 182832.066)	9.0
1	2	POINT (525208.067 179391.922)	27.0
2	3	POINT (532985.807 182001.572)	NaN
...	...	...	...
739	775	POINT (523391.016 179020.043)	NaN
740	776	POINT (527437.473 175077.168)	NaN
741	777	POINT (527553.301 175257)	NaN

The capacity of nearby stations can be verified by comparing a plot of the capacity of the source `cycle_hire_osm` data, with the join results in the new object `cycle_hire_buffers` (Figure 3.9).

```
# Input
fig, ax = plt.subplots(1, 1, figsize=(6, 3))
cycle_hire_osm.plot(column='capacity', legend=True, ax=ax);
# Join result
fig, ax = plt.subplots(1, 1, figsize=(6, 3))
cycle_hire_buffers.plot(column='capacity', legend=True, ax=ax);
```



(a) Input (`cycle_hire_osm`) (b) Join result (`cycle_hire_buffers`)

Figure 3.9: Non-overlapping join

3.2.5 Spatial aggregation

As with attribute data aggregation, spatial data aggregation condenses data: aggregated outputs have fewer rows than non-aggregated inputs. Statistical aggregating functions, such as mean, average, or sum, summarize multiple values of a variable, and return a single value per grouping variable. Section 2.2.2 demonstrated how the `.groupby` method, combined with summary functions such as `.sum`, condense data based on attribute variables. This section shows

how grouping by spatial objects can be achieved using spatial joins combined with non-spatial aggregation.

Returning to the example of New Zealand, imagine you want to find out the average height of `nz_height` points in each region. It is the geometry of the source (`nz`) that defines how values in the target object (`nz_height`) are grouped. This can be done in three steps:

1. Figuring out which `nz` region each `nz_height` point falls in—using `gpd.sjoin`
2. Summarizing the average elevation per region—using `.groupby` and `.mean`
3. Joining the result back to `nz`—using `pd.merge`

First, we 'attach' the region classification of each point, using spatial join (Section 3.2.3). Note that we are using the minimal set of columns required: the geometries (for the spatial join to work), the point elevation (to later calculate an average), and the region name (to use as key when joining the results back to `nz`). The result tells us which `nz` region each elevation point falls in.

```
nz_height2 = gpd.sjoin(
  nz_height[['elevation', 'geometry']],
  nz[['Name', 'geometry']],
  how='left'
)
nz_height2
```

	elevation	geometry	index_right	Name
0	2723	POINT (1204142.603 5049971.287)	12	Southland
1	2820	POINT (1234725.325 5048309.302)	11	Otago
2	2830	POINT (1235914.511 5048745.117)	11	Otago
...	...	...	...	...
98	2751	POINT (1820659.873 5649488.235)	2	Waikato
99	2720	POINT (1822262.592 5650428.656)	2	Waikato
100	2732	POINT (1822492.184 5650492.304)	2	Waikato

Second, we calculate the average elevation, using ordinary (non-spatial) aggregation (Section 2.2.2). This result tells us the average elevation of all `nz_height` points located within each `nz` region.

```
nz_height2 = nz_height2.groupby('Name')[['elevation']].mean().reset_index()
nz_height2
```

	Name	elevation
0	Canterbury	2994.600000
1	Manawatu-Wanganui	2777.000000
2	Marlborough	2720.000000
...	...	...
4	Southland	2723.000000
5	Waikato	2734.333333
6	West Coast	2889.454545

The third and final step is joining the averages back to the `nz` layer.

```
nz2 = pd.merge(nz[['Name', 'geometry']], nz_height2, on='Name', how='left')
nz2
```

	Name	geometry	elevation
0	Northland	MULTIPOLYGON (((1745493.196 600...	NaN
1	Auckland	MULTIPOLYGON (((1803822.103 590...	NaN
2	Waikato	MULTIPOLYGON (((1860345.005 585...	2734.333333
...	...	...	...
13	Tasman	MULTIPOLYGON (((1616642.877 542...	NaN
14	Nelson	MULTIPOLYGON (((1624866.278 541...	NaN
15	Marlborough	MULTIPOLYGON (((1686901.914 535...	2720.000000

We now have created the `nz2` layer, which gives the average `nz_height` elevation value per polygon. The result is shown in Figure 3.10. Note that the `missing_kwds` part determines the style of geometries where the symbology attribute (`elevation`) is missing, because there were no `nz_height` points overlapping with them. The default is to omit them, which is usually not what we want, but with `{'color':'grey','edgecolor':'black'}`, those polygons are shown with black outline and grey fill.

```
nz2.plot(
  column='elevation',
  legend=True,
  cmap='Blues', edgecolor='black',
  missing_kwds={'color': 'grey', 'edgecolor': 'black'}
);
```

3.2.6 Joining incongruent layers

Spatial congruence is an important concept related to spatial aggregation. An aggregating object (which we will refer to as `y`) is congruent with the target object (`x`) if the two objects have shared borders. Often this is the case for administrative boundary data, whereby larger units—such as Middle Layer Super Output Areas (MSOAs) in the UK, or districts in many other European countries—are composed of many smaller units.

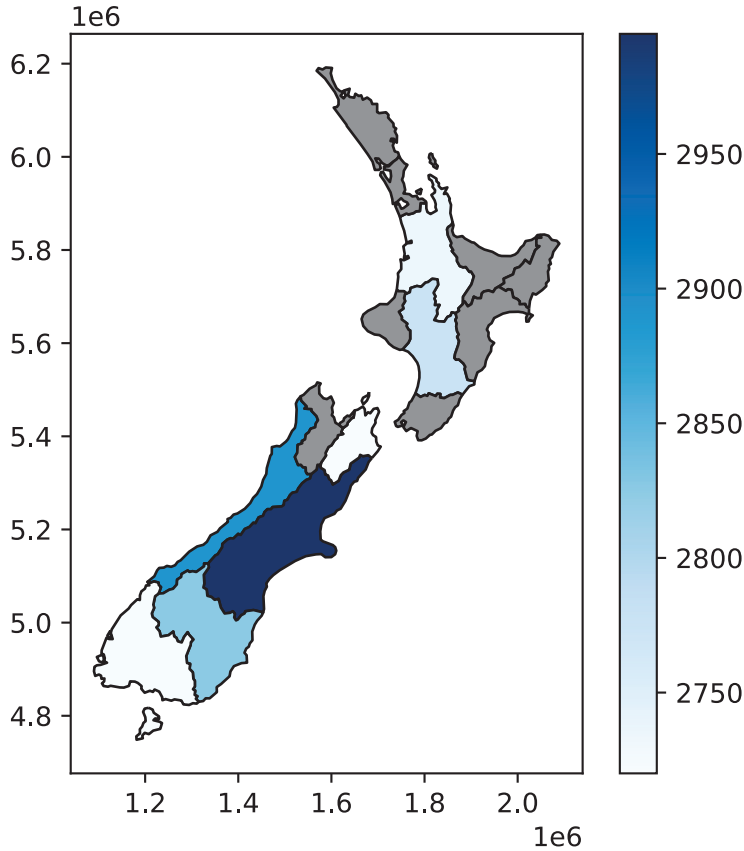


Figure 3.10: Average height of the top 101 high points across the regions of New Zealand

Incongruent aggregating objects, by contrast, do not share common borders with the target (Qiu, Zhang, and Zhou 2012). This is problematic for spatial aggregation (and other spatial operations) illustrated in Figure 3.11: aggregating the centroid of each sub-zone will not return accurate results. Areal interpolation overcomes this issue by transferring values from one set of areal units to another, using a range of algorithms including simple area-weighted approaches and more sophisticated approaches such as 'pycnophylactic' methods (Tobler 1979).

To demonstrate joining incongruent layers, we will create a 'synthetic' layer comprising a regular grid of rectangles of size 100×100 km, covering the extent of the `nz` layer. This recipe can be used to create a regular grid covering any given layer (other than `nz`), at the specified resolution (`res`). Most of the

functions have been explained in previous chapters; we leave it as an exercise for the reader to explore how the code works.

```
# Settings: grid extent, resolution, and CRS
bounds = nz.total_bounds
crs = nz.crs
res = 100000
# Calculating grid dimensions
xmin, ymin, xmax, ymax = bounds
cols = list(range(int(np.floor(xmin)), int(np.ceil(xmax+res)), res))
rows = list(range(int(np.floor(ymin)), int(np.ceil(ymax+res)), res))
rows.reverse()
# For each cell, create 'shapely' polygon (rectangle)
polygons = []
for x in cols:
    for y in rows:
        polygons.append(
            shapely.Polygon([(x,y), (x+res, y), (x+res, y-res), (x, y-res)])
        )
# To 'GeoDataFrame'
grid = gpd.GeoDataFrame({'geometry': polygons}, crs=crs)
# Remove rows/columns beyond the extent
sel = grid.intersects(shapely.box(*bounds))
grid = grid[sel]
# Add consecutive IDs
grid['id'] = grid.index
grid
```

	geometry	id
0	POLYGON ((1090143 6248536, 1190...	0
1	POLYGON ((1090143 6148536, 1190...	1
2	POLYGON ((1090143 6048536, 1190...	2
...	...	...
156	POLYGON ((1990143 5048536, 2090...	156
157	POLYGON ((1990143 4948536, 2090...	157
158	POLYGON ((1990143 4848536, 2090...	158

Figure 3.11 shows the newly created `grid` layer, along with the `nz` layer.

```
base = grid.plot(color='none', edgecolor='grey')
nz.plot(
    ax=base,
    column='Population',
    edgecolor='black',
    legend=True,
    cmap='Reds'
);
```

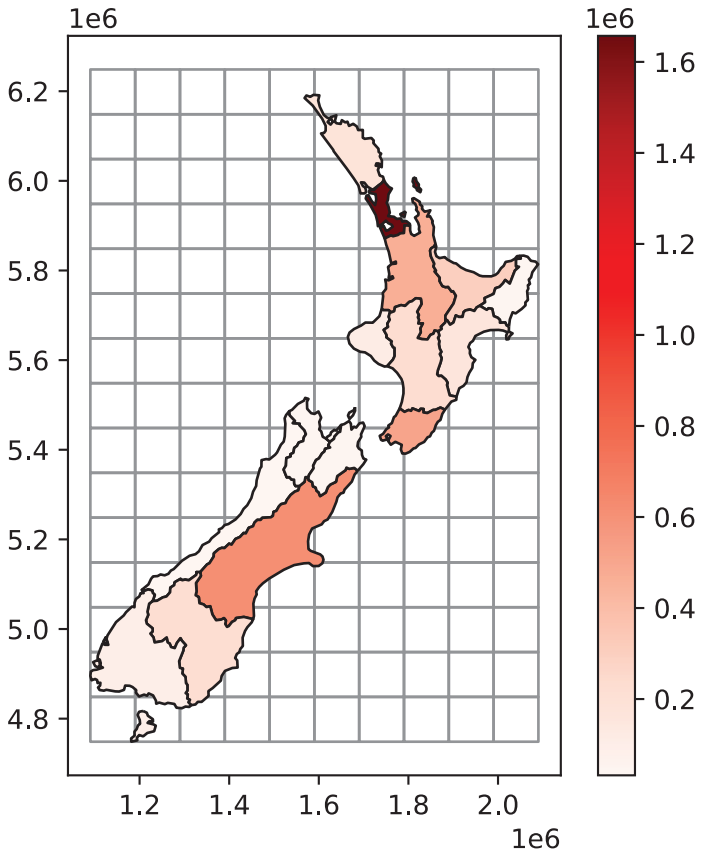


Figure 3.11: The `nz` layer, with population size in each region, overlaid with a regular `grid` of rectangles

Our goal, now, is to 'transfer' the `'Population'` attribute (Figure 3.11) to the rectangular grid polygons, which is an example of a join between incongruent layers. To do that, we basically need to calculate—for each `grid` cell—the weighted sum of the population in `nz` polygons coinciding with that cell. The weights in the weighted sum calculation are the ratios between the area of the coinciding 'part' out of the entire `nz` polygon. That is, we (inevitably) assume that the population in each `nz` polygon is equally distributed across space, therefore a partial `nz` polygon contains the respective partial population size.

We start by calculating the entire area of each `nz` polygon, as follows, using the `.area` method (Section 1.2.7).

```
nz['area'] = nz.area
nz
```

	Name	Island	...	geometry	area
0	Northland	North	...	MULTIPOLYGON (((1745493.196 600...	1.289058e+10
1	Auckland	North	...	MULTIPOLYGON (((1803822.103 590...	4.911565e+09
2	Waikato	North	...	MULTIPOLYGON (((1860345.005 585...	2.458882e+10
...	...	...	...	...	...
13	Tasman	South	...	MULTIPOLYGON (((1616642.877 542...	9.594918e+09
14	Nelson	South	...	MULTIPOLYGON (((1624866.278 541...	4.080754e+08
15	Marlborough	South	...	MULTIPOLYGON (((1686901.914 535...	1.046485e+10

Next, we use the `.overlay` method to calculate the pairwise intersections between `nz` and `grid`. As a result, we now have a layer where each `nz` polygon is split according to the `grid` polygons, hereby named `nz_grid`.

```
nz_grid = nz.overlay(grid)
nz_grid = nz_grid[['id', 'area', 'Population', 'geometry']]
nz_grid
```

	id	area	Population	geometry
0	64	1.289058e+10	175500.0	POLYGON ((1586362.965 6168009.0...
1	80	1.289058e+10	175500.0	POLYGON ((1590143 6162776.641, ...
2	81	1.289058e+10	175500.0	POLYGON ((1633099.964 6066188.0...
...	...	...	...	...
107	89	1.046485e+10	46200.0	POLYGON ((1641283.955 5341361.1...
108	103	1.046485e+10	46200.0	POLYGON ((1690724.332 5458875.4...
109	104	1.046485e+10	46200.0	MULTIPOLYGON (((1694233.995 543...

Figure 3.12 illustrates the effect of `.overlay`:

```
nz_grid.plot(color='none', edgecolor='black');
```

We also need to calculate the areas of the intersections, here into a new attribute `'area_sub'`. If an `nz` polygon was completely within a single `grid` polygon, then `area_sub` is going to be equal to `area`; otherwise, it is going to be smaller.

```
nz_grid['area_sub'] = nz_grid.area
nz_grid
```

	id	area	Population	geometry	area_sub
0	64	1.289058e+10	175500.0	POLYGON ((1586362.965 6168009.0...	3.231015e+08
1	80	1.289058e+10	175500.0	POLYGON ((1590143 6162776.641, ...	4.612641e+08
2	81	1.289058e+10	175500.0	POLYGON ((1633099.964 6066188.0...	5.685656e+09
...	...	...	...	...	...
107	89	1.046485e+10	46200.0	POLYGON ((1641283.955 5341361.1...	1.826943e+09

	id	area	Population	geometry	area_sub
108	103	1.046485e+10	46200.0	POLYGON ((1690724.332 5458875.4...	1.227037e+08
109	104	1.046485e+10	46200.0	MULTIPOLYGON (((1694233.995 543...	4.874611e+08

The resulting layer `nz_grid`, with the `area_sub` attribute, is shown in Figure 3.13.

```
base = grid.plot(color='none', edgecolor='grey')
nz_grid.plot(
    ax=base,
    column='area_sub',
    edgecolor='black',
    legend=True,
    cmap='Reds'
);
```

Note that each of the intersections still holds the `Population` attribute of its 'origin' feature of `nz`, i.e., each portion of the `nz` area is associated with the original complete population count for that area. The real population size of each `nz_grid` feature, however, is smaller, or equal, depending on the geographic area proportion that it occupies out of the original `nz` feature. To make the correction, we first calculate the ratio (`area_prop`) and then multiply it by the population. The new (lowercase) attribute `population` now has the correct estimate of population sizes in `nz_grid`:

```
nz_grid['area_prop'] = nz_grid['area_sub'] / nz_grid['area']
nz_grid['population'] = nz_grid['Population'] * nz_grid['area_prop']
nz_grid
```

	id	area	...	area_prop	population
0	64	1.289058e+10	...	0.025065	4398.897141
1	80	1.289058e+10	...	0.035783	6279.925114
2	81	1.289058e+10	...	0.441071	77407.916241
...	...	...	...	...	...
107	89	1.046485e+10	...	0.174579	8065.550415
108	103	1.046485e+10	...	0.011725	541.709946
109	104	1.046485e+10	...	0.046581	2152.033881

What is left to be done is to sum (see Section 2.2.2) the population in all parts forming the same grid cell and join (see Section 2.2.3) them back to the `grid` layer. Note that many of the grid cells have 'No Data' for population, because they have no intersection with `nz` at all (Figure 3.11).

```
nz_grid = nz_grid.groupby('id')['population'].sum().reset_index()
grid = pd.merge(grid, nz_grid[['id', 'population']], on='id', how='left')
grid
```

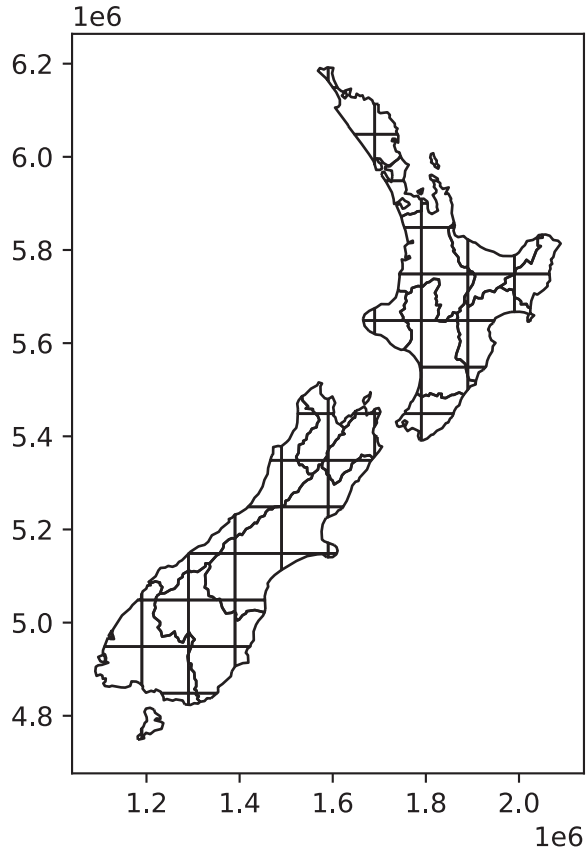


Figure 3.12: The pairwise intersections of `nz` and `grid`, calculated with `.overlay`

	geometry	id	population
0	POLYGON ((1090143 6248536, 1190...	0	NaN
1	POLYGON ((1090143 6148536, 1190...	1	NaN
2	POLYGON ((1090143 6048536, 1190...	2	NaN
...	...	...	...
147	POLYGON ((1990143 5048536, 2090...	156	NaN
148	POLYGON ((1990143 4948536, 2090...	157	NaN
149	POLYGON ((1990143 4848536, 2090...	158	NaN

Figure 3.14 shows the final result `grid` with the incongruently-joined `population` attribute from `nz`.

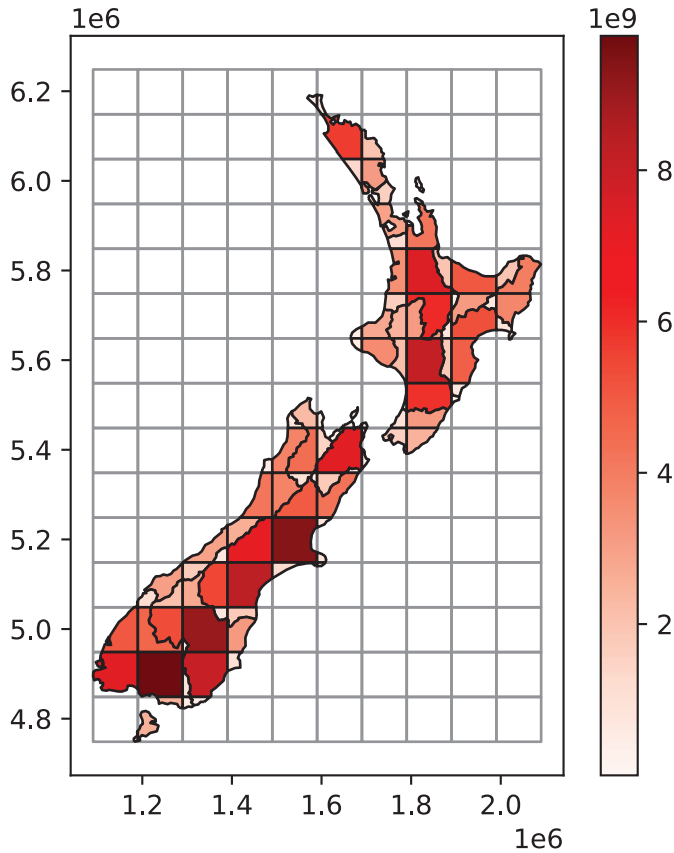


Figure 3.13: The areas of pairwise intersections in the `nz_grid` layer

```
base = grid.plot(
    column='population',
    edgecolor='black',
    legend=True,
    cmap='Reds'
);
nz.plot(ax=base, color='none', edgecolor='grey', legend=True);
```

We can demonstrate that, expectedly, the summed population in `nz` and `grid` is identical, even though the geometry is different (since we created `grid` to completely cover `nz`), by comparing the `.sum` of the `population` attribute in both layers.

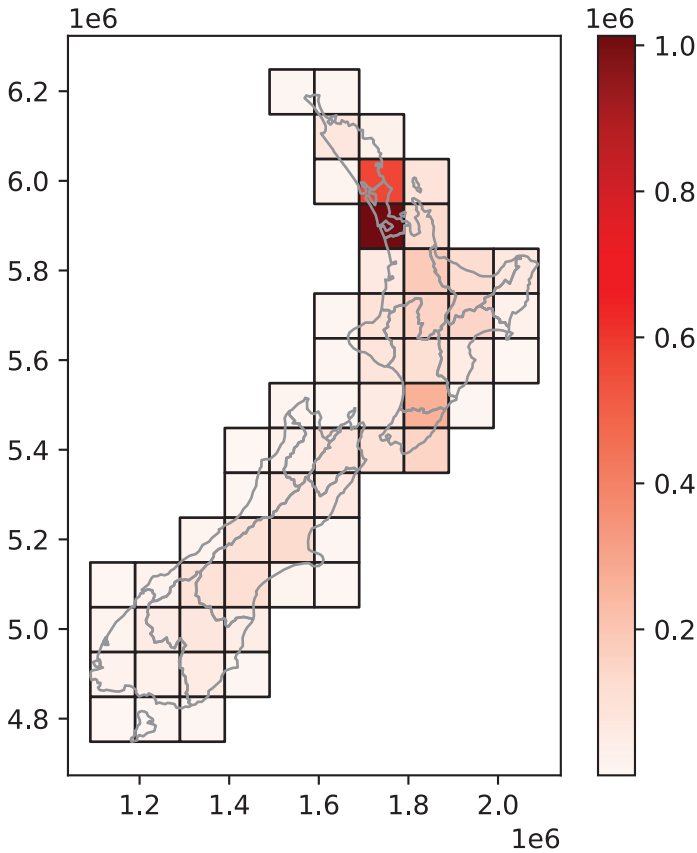


Figure 3.14: The `nz` layer and a regular grid of rectangles: final result

```
nz['Population'].sum()
```

```
np.float64(4787200.0)
```

```
grid['population'].sum()
```

```
np.float64(4787199.999999998)
```

The procedure in this section is known as an area-weighted interpolation of a spatially *extensive* (e.g., population) variable. In extensive interpolation, we assume that the variable of interest represents counts (such as, here, inhabitants) uniformly distributed across space. In such case, each part of a given polygon captures the respective proportion of counts (such as, half of a region with N inhabitants contains $N/2$ inhabitants). Accordingly, summing the parts gives the total count of the total area.

An area-weighted interpolation of a spatially *intensive* variable (e.g., population density) is almost identical, except that we would have to calculate the weighted `.mean` rather than `.sum`, to preserve the average rather than the sum. In intensive interpolation, we assume that the variable of interest represents counts per unit area, i.e., density. Since density is (assumed to be) uniform, any part of a given polygon has exactly the same density as that of the whole polygon. Density values are therefore computed as weighted averages, rather than sums, of the parts. Also, see the 'Area-weighted interpolation' section in Pebesma and Bivand (2023).

3.2.7 Distance relations

While topological relations are binary—a feature either intersects with another or does not—distance relations are continuous. The distance between two objects is calculated with the `.distance` method. The method is applied on a `GeoSeries` (or a `GeoDataFrame`), with the argument being an individual `shapely` geometry. The result is a `Series` of pairwise distances.

> **Note**
>
> **geopandas** uses similar syntax and mode of operation for many of its methods and functions, including:
> - Numeric calculations, such as `.distance` (this section), returning numeric values
> - Topological evaluation methods, such as `.intersects` or `.disjoint` (Section 3.2.2), returning boolean values
> - Geometry generating-methods, such as `.intersection` (Section 4.2.5), returning geometries
>
> In all cases, the input is a `GeoSeries` and (or a `GeoDataFrame`) and a `shapely` geometry, and the output is a `Series` or `GeoSeries` of results, contrasting each geometry from the `GeoSeries` with the `shapely` geometry. The examples in this book demonstrate this, so-called 'many-to-one', mode of the functions.
> All of the above-mentioned methods also have a pairwise mode, perhaps less useful and not used in the book, where we evaluate relations between pairs of geometries in two `GeoSeries`, aligned either by index or by position.

To illustrate the `.distance` method, let's take the three highest points in New Zealand with `.sort_values` and `.iloc`.

```
nz_highest = nz_height.sort_values(by='elevation', ascending=False).iloc[:3, :]
nz_highest
```

	t50_fid	elevation	geometry
64	2372236	3724	POINT (1369317.63 5169132.284)
63	2372235	3717	POINT (1369512.866 5168235.616)
67	2372252	3688	POINT (1369381.942 5168761.875)

Additionally, we need the geographic centroid of the Canterbury region (`canterbury`, created in Section 3.2.1).

```
canterbury_centroid = canterbury.centroid.iloc[0]
```

Now we are able to apply `.distance` to calculate the distances from each of the three elevation points to the centroid of the Canterbury region.

```
nz_highest.distance(canterbury_centroid)
```

```
64    115539.995747
63    115390.248038
67    115493.594066
dtype: float64
```

To obtain a distance matrix, i.e., a pairwise set of distances between all combinations of features in objects `x` and `y`, we need to use the `.apply` method (analogous to the way we created the `.intersects` boolean matrix in Section 3.2.2). To illustrate this, let's now take two regions in `nz`, Otago and Canterbury, represented by the object `co`.

```
sel = nz['Name'].str.contains('Canter|Otag')
co = nz[sel]
co
```

	Name	Island	...	geometry	area
10	Canterbury	South	...	MULTIPOLYGON (((1686901.914 535...	4.532656e+10
11	Otago	South	...	MULTIPOLYGON (((1335204.789 512...	3.190356e+10

The distance matrix (technically speaking, a `DataFrame`) `d` between each of the first three elevation points, and the two regions, is then obtained as follows. In plain language, we take the geometry from each row in `nz_height.iloc[:3,:]`, and apply the `.distance` method on `co` with its rows as the argument.

```
d = nz_height.iloc[:3, :].apply(lambda x: co.distance(x.geometry), axis=1)
d
```

	10	11
0	123537.158269	15497.717252
1	94282.773074	0.000000
2	93018.560814	0.000000

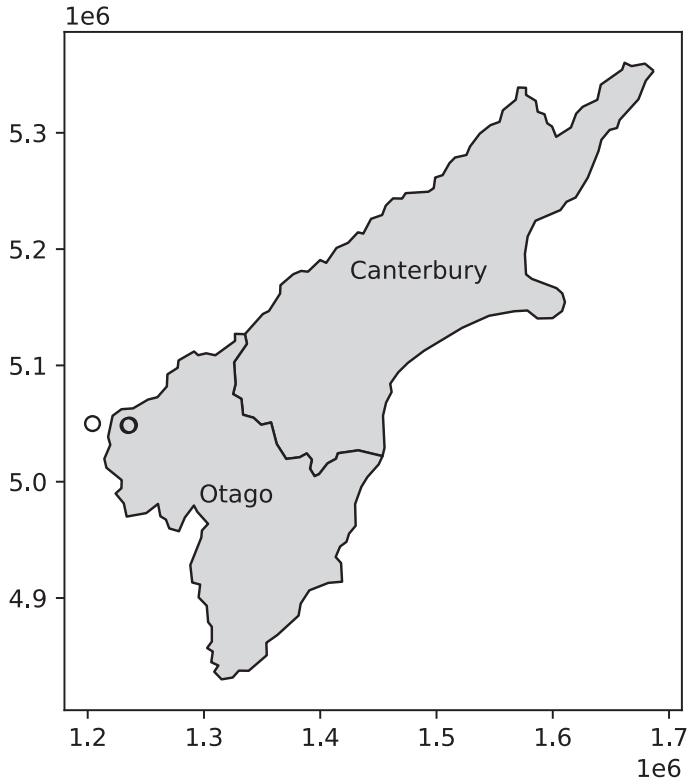


Figure 3.15: The first three `nz_height` points, and the Otago and Canterbury regions from `nz`

Note that the distance between the second and third features in `nz_height` and the second feature in `co` is zero. This demonstrates the fact that distances between points and polygons refer to the distance to any part of the polygon: the second and third points in `nz_height` are in Otago, which can be verified by plotting them (two almost completely overlapping points in Figure 3.15).

```
fig, ax = plt.subplots()
co.plot(color='lightgrey', edgecolor='black', ax=ax)
co.apply(
    lambda x: ax.annotate(
        text=x['Name'],
        xy=x.geometry.centroid.coords[0],
        ha='center'
    ),
    axis=1
)
nz_height.iloc[:3, :].plot(color='none', edgecolor='black', ax=ax);
```

3.3 Spatial operations on raster data

This section builds on Section 2.3, which highlights various basic methods for manipulating raster datasets, to demonstrate more advanced and explicitly spatial raster operations, and uses the `elev.tif` and `grain.tif` rasters manually created in Section 1.3.2.

3.3.1 Spatial subsetting

The previous chapter (and especially Section 2.3) demonstrated how to retrieve values associated with specific row and column combinations from a raster. Raster values can also be extracted by location (coordinates) and other spatial objects. To use coordinates for subsetting, we can use the `.sample` method of a `rasterio` file connection object, combined with a list of coordinate tuples. The method is demonstrated below to find the value of the cell that covers a point located at coordinates of `(0.1,0.1)` in `elev`. The returned object is a *generator*. The rationale for returning a generator, rather than a `list`, is memory efficiency. The number of sampled points may be huge, in which case we would want to generate the values one at a time rather than all at once.

```
src_elev.sample([(0.1, 0.1)])
```

```
<generator object sample_gen at 0x7f43f83101c0>
```

> **Note**
>
> The technical terms *iterable*, *iterator*, and *generator* in Python may be confusing, so here is a short summary, ordered from most general to most specific:
>
> - An *iterable* is any object that we can iterate on, such as using a `for` loop. For example, a `list` is iterable.
> - An *iterator* is an object that represents a stream of data, which we can go over, each time getting the next element using `next`. Iterators are also iterable, meaning that you can over them in a loop, but they are stateful (e.g., they remember which item was obtained using `next`), meaning that you can go over them just once.
> - A *generator* is a function that returns an iterator. For example, the `.sample` method in the above example is a generator. The **rasterio** package makes use of generators in some of its functions, as we will see later on (Section 5.5.1).

In case we nevertheless want all values at once, such as when the number of points is small, we can force the generation of all values from a generator at

once, using `list`. Since there was just one point, the result is one extracted value, in this case `16`.

```
list(src_elev.sample([(0.1, 0.1)]))
```

```
[array([16], dtype=uint8)]
```

We can use the same technique to extract the values of multiple points at once. For example, here we extract the raster values at two points, `(0.1,0.1)` and `(1.1,1.1)`. The resulting values are `16` and `6`.

```
list(src_elev.sample([(0.1, 0.1), (1.1, 1.1)]))
```

```
[array([16], dtype=uint8), array([6], dtype=uint8)]
```

The location of the two sample points on top of the `elev.tif` raster is illustrated in Figure 3.16.

```
fig, ax = plt.subplots()
rasterio.plot.show(src_elev, ax=ax)
gpd.GeoSeries([shapely.Point(0.1, 0.1)]) \
    .plot(color='black', edgecolor='white', markersize=50, ax=ax)
gpd.GeoSeries([shapely.Point(1.1, 1.1)]) \
    .plot(color='black', edgecolor='white', markersize=50, ax=ax);
```

> **Note**
>
> We elaborate on the plotting technique used to display the points and the raster in Section 8.2.5. We will also introduce a more user-friendly and general method to extract raster values to points, using the **rasterstats** package, in Section 5.3.1.

Another common use case of spatial subsetting is using a boolean mask, based on another raster with the same extent and resolution, or the original one, as illustrated in Figure 3.17. To do that, we erase the values in the array of one raster, according to another corresponding mask raster. For example, let's read (Section 1.3.1) the `elev.tif` raster values into an array named `elev` (Figure 3.17 (a)).

```
elev = src_elev.read(1)
elev
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35, 36]], dtype=uint8)
```

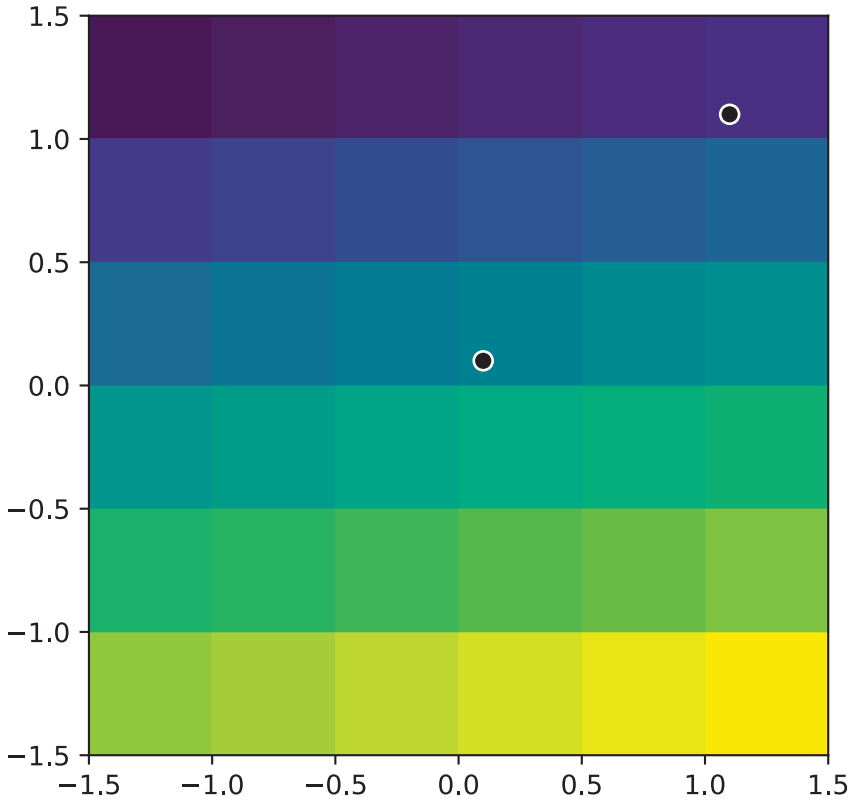


Figure 3.16: The `elev.tif` raster, and two points where we extract its values

and create a corresponding random boolean mask named `mask` (Figure 3.17 (b)), of the same shape as `elev.tif` with values randomly assigned to `True` and `False`.

```
np.random.seed(1)
mask = np.random.choice([True, False], src_elev.shape)
mask
```

```
array([[False, False,  True,  True, False, False],
       [False, False, False,  True,  True, False],
       [ True, False, False,  True,  True, False],
       [ True,  True,  True, False,  True,  True],
       [False,  True,  True,  True, False,  True],
       [ True,  True, False, False, False, False]])
```

Next, suppose that we want to keep only those values of `elev` which are `False` in `mask` (i.e., they are *not* masked). In other words, we want to mask `elev` with

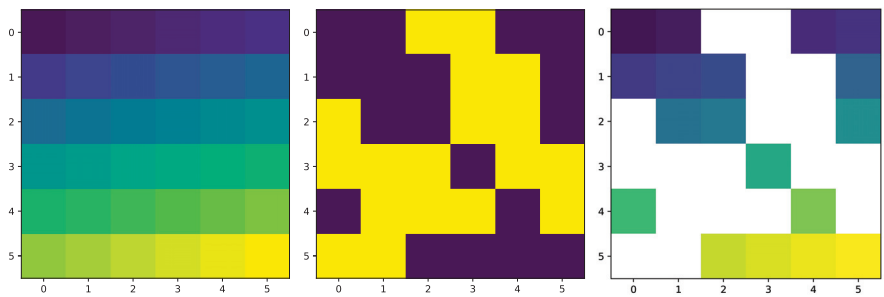
`mask`. The result will be stored in a copy named `masked_elev` (Figure 3.17 (c)). In the case of `elev.tif`, to be able to store `np.nan` in the array of values, we also need to convert it to `float` (see Section 2.3.2). Afterwards, masking is a matter of assigning `np.nan` into a subset defined by the mask, using the 'boolean array indexing' syntax of **numpy**.

```
masked_elev = elev.copy()
masked_elev = masked_elev.astype('float64')
masked_elev[mask] = np.nan
masked_elev
```

```
array([[ 1.,  2., nan, nan,  5.,  6.],
       [ 7.,  8.,  9., nan, nan, 12.],
       [nan, 14., 15., nan, nan, 18.],
       [nan, nan, nan, 22., nan, nan],
       [25., nan, nan, nan, 29., nan],
       [nan, nan, 33., 34., 35., 36.]])
```

Figure 3.17 shows the original `elev` raster, the `mask` raster, and the resulting `masked_elev` raster.

```
rasterio.plot.show(elev);
rasterio.plot.show(mask);
rasterio.plot.show(masked_elev);
```



(a) Original raster (b) Raster mask (c) Output masked raster

Figure 3.17: Subsetting raster values using a boolean mask

The mask can be created from the array itself, using condition(s). That way, we can replace some values (e.g., values assumed to be wrong) with `np.nan`, such as in the following example.

```
elev2 = elev.copy()
elev2 = elev2.astype('float64')
elev2[elev2 < 20] = np.nan
elev2
```

```
array([[nan, nan, nan, nan, nan, nan],
       [nan, nan, nan, nan, nan, nan],
       [nan, nan, nan, nan, nan, nan],
       [nan, 20., 21., 22., 23., 24.],
       [25., 26., 27., 28., 29., 30.],
       [31., 32., 33., 34., 35., 36.]])
```

This technique is also used to reclassify raster values (see Section 3.3.3).

3.3.2 Map algebra

The term 'map algebra' was coined in the late 1970s to describe a 'set of conventions, capabilities, and techniques' for the analysis of geographic raster and (although less prominently) vector data (Tomlin 1994). In this context, we define map algebra more narrowly, as operations that modify or summarize raster cell values, with reference to surrounding cells, zones, or statistical functions that apply to every cell.

Map algebra operations tend to be fast, because raster datasets only implicitly store coordinates, hence the old adage 'raster is faster but vector is corrector'. The location of cells in raster datasets can be calculated by using its matrix position and the resolution and origin of the dataset (stored in the raster metadata, Section 1.3.1). For the processing, however, the geographic position of a cell is barely relevant as long as we make sure that the cell position is still the same after the processing. Additionally, if two or more raster datasets share the same extent, projection, and resolution, one could treat them as matrices for the processing.

Map algebra (or cartographic modeling with raster data) divides raster operations into four subclasses (Tomlin 1990), with each working on one or several grids simultaneously:

- Local or per-cell operations (Section 3.3.3)
- Focal or neighborhood operations. Most often the output cell value is the result of a 3×3 input cell block (Section 3.3.4)
- Zonal operations are similar to focal operations, but the surrounding pixel grid on which new values are computed can have irregular sizes and shapes (Section 3.3.5)
- Global or per-raster operations; that means the output cell derives its value potentially from one or several entire rasters (Section 3.3.6)

This typology classifies map algebra operations by the number of cells used for each pixel processing step and the type of output. For the sake of completeness, we should mention that raster operations can also be classified by disciplines such as terrain, hydrological analysis, or image classification. The following sections explain how each type of map algebra operations can be used, with reference to worked examples.

3.3.3 Local operations

Local operations comprise all cell-by-cell operations in one or several layers. Raster algebra is a classical use case of local operations—this includes adding or subtracting values from a raster, squaring, and multiplying rasters. Raster algebra also allows logical operations such as finding all raster cells that are greater than a specific value (e.g., `5` in our example below). Local operations are applied using the **numpy** array operations syntax, as demonstrated below.

First, let's take the array of `elev.tif` raster values, which we already read earlier (Section 3.3.1).

```
elev
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35, 36]], dtype=uint8)
```

Now, any element-wise array operation can be applied using **numpy** arithmetic or conditional operators and functions, comprising local raster operations in spatial analysis terminology. For example, `elev+elev` adds the values of `elev` to itself, resulting in a raster with double values.

```
elev + elev
```

```
array([[ 2,  4,  6,  8, 10, 12],
       [14, 16, 18, 20, 22, 24],
       [26, 28, 30, 32, 34, 36],
       [38, 40, 42, 44, 46, 48],
       [50, 52, 54, 56, 58, 60],
       [62, 64, 66, 68, 70, 72]], dtype=uint8)
```

Note that some functions and operators automatically change the data type to accommodate the resulting values, while other operators do not, potentially resulting in overflow (i.e., incorrect values for results beyond the data type range, such as trying to accommodate values above `255` in an `int8` array). For example, `elev**2` (`elev` squared) results in overflow. Since the `**` operator does not automatically change the data type, leaving it as `int8`, the resulting array has incorrect values for `16**2`, `17**2`, etc., which are above `255` and therefore cannot be accommodated.

```
elev**2
```

```
array([[  1,   4,   9,  16,  25,  36],
       [ 49,  64,  81, 100, 121, 144],
       [169, 196, 225,   0,  33,  68],
       [105, 144, 185, 228,  17,  64],
       [113, 164, 217,  16,  73, 132],
       [193,   0,  65, 132, 201,  16]], dtype=uint8)
```

To avoid this situation, we can, for instance, transform `elev` to the standard `int64` data type, using `.astype` before applying the `**` operator. That way, all results, up to `36**2` (`1296`), can be easily accommodated, since the `int64` data type supports values up to `9223372036854775807` (Table 7.2).

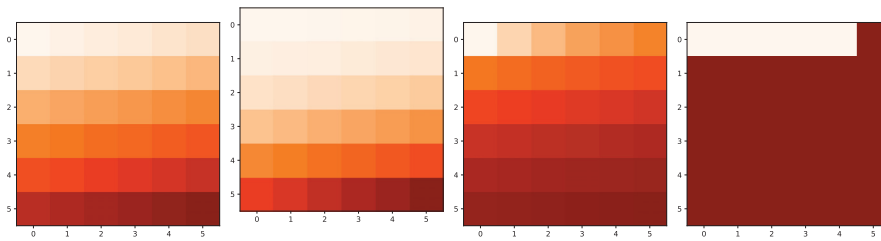
```
elev.astype(int)**2
```

```
array([[   1,    4,    9,   16,   25,   36],
       [  49,   64,   81,  100,  121,  144],
       [ 169,  196,  225,  256,  289,  324],
       [ 361,  400,  441,  484,  529,  576],
       [ 625,  676,  729,  784,  841,  900],
       [ 961, 1024, 1089, 1156, 1225, 1296]])
```

Now we get correct results.

Figure 3.18 demonstrates the result of the last two examples (`elev+elev` and `elev.astype(int)**2`), and two other ones (`np.log(elev)` and `elev>5`).

```
rasterio.plot.show(elev + elev, cmap='Oranges');
rasterio.plot.show(elev.astype(int)**2, cmap='Oranges');
rasterio.plot.show(np.log(elev), cmap='Oranges');
rasterio.plot.show(elev > 5, cmap='Oranges');
```



(a) `elev+elev` (b) `elev.astype (int)**2` (c) `np.log(elev)` (d) `elev>5`

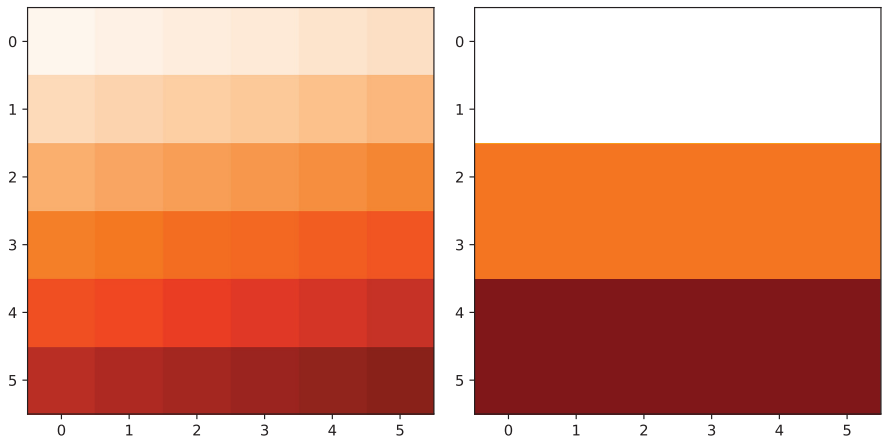
Figure 3.18: Examples of different local operations of the elev raster object: adding two rasters, squaring, applying logarithmic transformation, and performing a logical operation.

Another good example of local operations is the classification of intervals of numeric values into groups such as grouping a digital elevation model into low (class 1), middle (class 2) and high (class 3) elevations. Here, the raster values in the ranges 0–12, 12–24, and 24–36 are reclassified to take values 1, 2, and 3, respectively.

```
recl = elev.copy()
recl[(elev > 0)  & (elev <= 12)] = 1
recl[(elev > 12) & (elev <= 24)] = 2
recl[(elev > 24) & (elev <= 36)] = 3
```

Figure 3.19 compares the original `elev` raster with the reclassified `recl` one.

```
rasterio.plot.show(elev, cmap='Oranges');
rasterio.plot.show(recl, cmap='Oranges');
```



(a) Original (b) Reclassified

Figure 3.19: Reclassifying a continuous raster into three categories.

The calculation of the Normalized Difference Vegetation Index (NDVI)[2] is a well-known local (pixel-by-pixel) raster operation. It returns a raster with values between `-1` and `1`; positive values indicate the presence of living plants (mostly $>$ `0.2`). NDVI is calculated from red and near-infrared (NIR) bands of remotely sensed imagery, typically from satellite systems such as Landsat or Sentinel-2. Vegetation absorbs light heavily in the visible light spectrum, and especially in the red channel, while reflecting NIR light, which is emulated in the NVDI formula (Equation 3.1),

$$NDVI = \frac{NIR - Red}{NIR + Red} \tag{3.1}$$

, where NIR is the near-infrared band and Red is the red band.

Let's calculate NDVI for the multispectral Landsat satellite file (`landsat.tif`) of the Zion National Park. The file `landsat.tif` contains surface reflectance values (range `0-1`) in the blue, green, red, and near-infrared (NIR) bands. We start by reading the file and extracting the NIR and red bands, which are the fourth and third bands, respectively. Next, we apply the formula to calculate the NDVI values.

[2] https://en.wikipedia.org/wiki/Normalized_difference_vegetation_index

```
landsat = src_landsat.read()
nir = landsat[3]
red = landsat[2]
ndvi = (nir-red)/(nir+red)
```

When plotting an RGB image using the `rasterio.plot.show` function, the function assumes that values are in the range `[0,1]` for floats, or `[0,255]` for integers (otherwise clipped) and the order of bands is RGB. To prepare the multi-band raster for `rasterio.plot.show`, we, therefore, reverse the order of the first three bands (to go from B-G-R-NIR to R-G-B), using the `[:3]` slice to select the first three bands and then the `[::-1]` slice to reverse the bands order, and divide by the raster maximum to set the maximum value to `1`.

```
landsat_rgb = landsat[:3][::-1] / landsat.max()
```

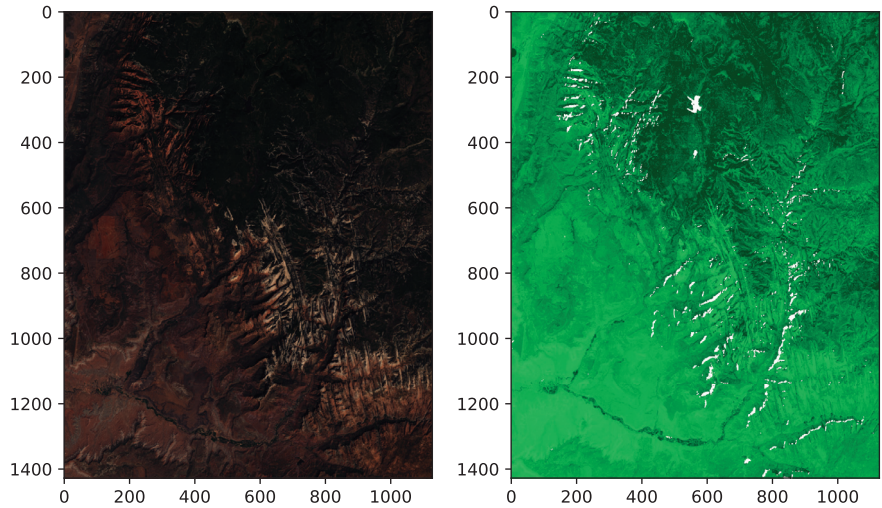
> **Note**
>
> Python slicing notation, which **numpy**, **pandas** and **geopandas** also follow, is `object[start:stop:step]`. The default is to start from the beginning, go to the end, and use steps of `1`. Otherwise, `start` is inclusive and `end` is exclusive, whereas negative `step` values imply going backwards starting from the end. Also, always keep in mind that Python indices start from `0`. When subsetting two- or three-dimensional objects, indices for each dimension are separated by commas, where either index can be set to `:` meaning 'all values'. The last dimensions can also be omitted implying `:`, e.g., to subset the first three bands from a three-dimensional array `a` we can use either `a[:3,:,:]` or `a[:3]`.
> In the above example:
> - The slicing expression `[:3]` therefore means layers `0`, `1`, `2` (up to `3`, exclusive)
> - The slicing expression `[::-1]` therefore means all (three) bands in reverse order

Figure 3.20 shows the RGB image and the NDVI values calculated for the Landsat satellite image of the Zion National Park.

```
rasterio.plot.show(landsat_rgb, cmap='RdYlGn');
rasterio.plot.show(ndvi, cmap='Greens');
```

3.3.4 Focal operations

While local functions operate on one cell at a time (though possibly from multiple layers), focal operations take into account a central (focal) cell and its neighbors. The neighborhood (also named kernel, filter, or moving window) under consideration is typically of 3×3 cells (that is, the central cell and its



(a) RGB image (b) NDVI

Figure 3.20: RGB image and NDVI values calculated for the Landsat satellite image of the Zion National Park

eight surrounding neighbors), but can take on any other (not necessarily rectangular) shape as defined by the user. A focal operation applies an aggregation function to all cells within the specified neighborhood, uses the corresponding output as the new value for the central cell, and moves on to the next central cell (Figure 3.21). Other names for this operation are spatial filtering and convolution (Burrough, McDonnell, and Lloyd 2015).

In Python, the **scipy.ndimage** (Virtanen et al. 2020) package has a comprehensive collection of functions to perform filtering of **numpy** arrays, such as:

- `scipy.ndimage.minimum_filter`,
- `scipy.ndimage.maximum_filter`,
- `scipy.ndimage.uniform_filter` (i.e., mean filter),
- `scipy.ndimage.median_filter`, etc.

In this group of functions, we define the shape of the moving window with either one of `size`—a single number (e.g., `3`), or tuple (e.g., `(3,3)`), implying a filter of those dimensions, or `footprint`—a boolean array, representing both the window shape and the identity of elements being included.

In addition to specific built-in filters, `convolve`—applies the sum function after multiplying by a custom `weights` array, and `generic_filter`—makes it possible to pass any custom function, where the user can specify any type of custom window-based calculation.

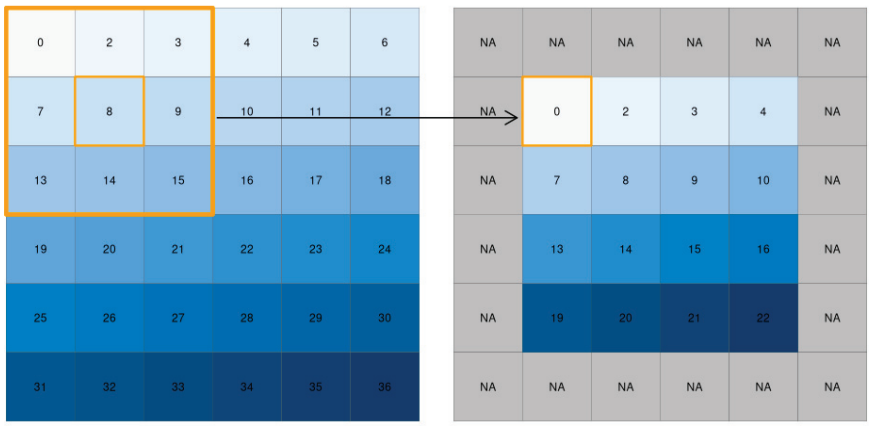


Figure 3.21: Input raster (left) and resulting output raster (right) due to a focal operation—finding the minimum value in 3×3 moving windows.

For example, here we apply the minimum filter with window size of `3` on `elev`. As a result, we now have a new array `elev_min`, where each value is the minimum in the corresponding 3×3 neighborhood in `elev`.

```
elev_min = scipy.ndimage.minimum_filter(elev, size=3)
elev_min
```

```
array([[ 1,  1,  2,  3,  4,  5],
       [ 1,  1,  2,  3,  4,  5],
       [ 7,  7,  8,  9, 10, 11],
       [13, 13, 14, 15, 16, 17],
       [19, 19, 20, 21, 22, 23],
       [25, 25, 26, 27, 28, 29]], dtype=uint8)
```

Special care should be given to the edge pixels – how should they be calculated? The **scipy.ndimage** filtering functions give several options through the `mode` parameter (see the documentation of any filtering function, such as `scipy.ndimage.median_filter`, for the definition of each mode): `reflect` (the default), `constant`, `nearest`, `mirror`, `wrap`. Sometimes artificially extending raster edges is considered unsuitable. In other words, we may wish the resulting raster to contain pixel values with 'complete' windows only, for example, to have a uniform sample size or because values in all directions matter (such as in topographic calculations). There is no specific option *not* to extend edges in **scipy.ndimage**. However, to get the same effect, the edges of the filtered array can be assigned with `np.nan`, in a number of rows and columns according to filter size. For example, when using a filter of `size=3`,

the outermost 'layer' of pixels may be assigned with `np.nan`, reflecting the fact that these pixels have incomplete 3×3 neighborhoods (Figure 3.21):

```
elev_min = elev_min.astype(float)
elev_min[:, [0, -1]] = np.nan
elev_min[[0, -1], :] = np.nan
elev_min
```

```
array([[nan, nan, nan, nan, nan, nan],
       [nan,  1.,  2.,  3.,  4., nan],
       [nan,  7.,  8.,  9., 10., nan],
       [nan, 13., 14., 15., 16., nan],
       [nan, 19., 20., 21., 22., nan],
       [nan, nan, nan, nan, nan, nan]])
```

We can quickly check if the output meets our expectations. In our example, the minimum value has to be always the upper left corner of the moving window (remember we have created the input raster by row-wise incrementing the cell values by one, starting at the upper left corner).

Focal functions or filters play a dominant role in image processing. For example, low-pass or smoothing filters use the mean function to remove extremes. By contrast, high-pass filters, often created with custom neighborhood weights, accentuate features.

In the case of categorical data, we can replace the mean with the mode, i.e., the most common value. To demonstrate applying a mode filter, let's read the small sample categorical raster `grain.tif`.

```
grain = src_grain.read(1)
grain
```

```
array([[1, 0, 1, 2, 2, 2],
       [0, 2, 0, 0, 2, 1],
       [0, 2, 2, 0, 0, 2],
       [0, 0, 1, 1, 1, 1],
       [1, 1, 1, 2, 1, 1],
       [2, 1, 2, 2, 0, 2]], dtype=uint8)
```

There is no built-in filter function for a mode filter in **scipy.ndimage**, but we can use the `scipy.ndimage.generic_filter` function along with a custom filtering function, internally utilizing `scipy.stats.mode`.

```
grain_mode = scipy.ndimage.generic_filter(
    grain,
    lambda x: scipy.stats.mode(x.flatten())[0],
    size=3
)
grain_mode = grain_mode.astype(float)
grain_mode[:, [0, -1]] = np.nan
```

```
grain_mode[[0, -1], :] = np.nan
grain_mode
```

```
array([[nan, nan, nan, nan, nan, nan],
       [nan,  0.,  0.,  0.,  2., nan],
       [nan,  0.,  0.,  0.,  1., nan],
       [nan,  1.,  1.,  1.,  1., nan],
       [nan,  1.,  1.,  1.,  1., nan],
       [nan, nan, nan, nan, nan, nan]])
```

> **Note**
>
> `scipy.stats.mode` is a function to summarize array values, returning the mode (most common value). It is analogous to **numpy** summary functions and methods, such as `.mean` or `.max`. **numpy** itself does not provide the *mode* function, however, which is why we use **scipy** for that.

Terrain processing is another important application of focal operations. Such functions are provided by multiple Python packages, including the general purpose **xarray** package, and more specialized packages such as **richdem** and **pysheds**. Useful terrain metrics include:

- Slope, measured in units of percent, degrees, or radians (Horn 1981)
- Aspect, meaning each cell's downward slope direction (Horn 1981)
- Slope curvature, including 'planform' and 'profile' curvature (Zevenbergen and Thorne 1987)

For example, each of these, and other, terrain metrics can be computed with the **richdem** package.

> **Note**
>
> Terrain metrics are essentially focal filters with customized functions. Using `scipy.ndimage.generic_filter`, along with such custom functions, is an option for those who would like to calculate terrain metric through coding by hand and/or limiting their code dependencies. For example, the *How Aspect works*[3] and *How Slope works*[4] pages from the ArcGIS Pro documentation provide explanations and formulas of the required functions for aspect and slope metrics (Figure 3.22), respectively, which can be translated to **numpy**-based functions to be used in `scipy.ndimage.generic_filter` to calculate those metrics.

[3] https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-analyst/how-aspect-works.htm

[4] https://pro.arcgis.com/en/pro-app/latest/tool-reference/spatial-analyst/how-slope-works.htm

Another extremely fast, memory-efficient, and concise, alternative, is to the use the GDAL program called `gdaldem`. `gdaldem` can be used to calculate slope, aspect, and other terrain metrics through a single command, accepting an input file path and exporting the result to a new file. This is our first example in the book where we demonstrate a situation where it may be worthwhile to leave the Python environment, and utilize a GDAL program directly, rather than through their wrappers (such as **rasterio** and other Python packages), whether to access a computational algorithm not easily accessible in a Python package, or for GDAL's memory-efficiency and speed benefits.

> **Note**
>
> GDAL contains a collection of over 40 programs, mostly aimed at raster processing. These include programs for fundamental operations, such as:
> - `gdal_translate`—convert between raster file formats
> - `gdalwarp`—raster reprojection
> - `gdal_rasterize`—rasterize vector features
> - `gdal_merge.py`—raster mosaic
>
> In this book, we use **rasterio** for the above-mentioned operations, although the GDAL programs are a good alternative for those who are more comfortable with the command line. However, we do use two GDAL programs for tasks that are lacking in **rasterio** and not well-implemented in other Python packages: `gdaldem` (this section), and `gdal_contour` (Section 5.5.3).

GDAL, along with all of its programs, should be available in your Python environment, since GDAL is a dependency of **rasterio**. The following example, which should be run from the command line, takes the `srtm_32612.tif` raster (which we are going to create in Section 6.8, therefore it is in the `'output'` directory), calculates slope (in decimal degrees, between `0` and `90`), and exports the result to a new file `srtm_32612_slope.tif`. Note that the arguments of `gdaldem` are the metric name (`slope`), then the input file path, and finally the output file path.

```
os.system('gdaldem slope output/srtm_32612.tif output/srtm_32612_slope.tif')
```

Here we ran the `gdaldem` command through `os.system`, in order to remain in the Python environment, even though we are calling an external program. Alternatively, you can run the standalone command in the command line interface you are using, such as the Anaconda Prompt:

```
gdaldem slope output/srtm_32612.tif output/srtm_32612_slope.tif
```

Replacing the metric name, we can calculate other terrain properties. For example, here is how we can calculate an aspect raster `srtm_32612_aspect.tif`, also in degrees (between `0` and `360`).

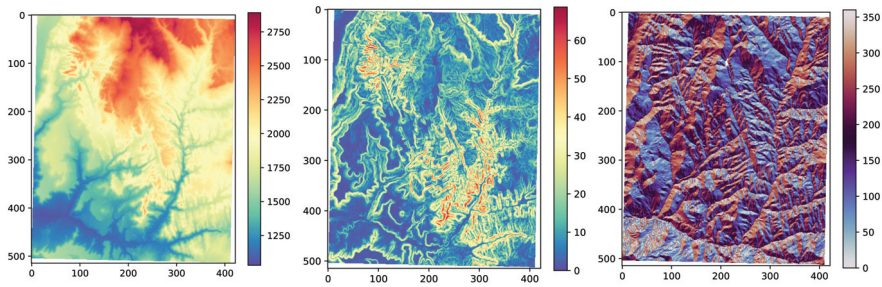
```
os.system('gdaldem aspect output/srtm_32612.tif output/srtm_32612_aspect.tif')
```

Figure 3.22 shows the results, using our more familiar plotting methods from **rasterio**. The code section is relatively long due to the workaround to create a color key (see Section 8.2.3) and removing 'No Data' flag values from the arrays so that the color key does not include them. Also note that we are using one of **matplotlib**'s cyclic color scales (`'twilight'`) when plotting aspect (Figure 3.22 (c)).

```
# Input DEM
src_srtm = rasterio.open('output/srtm_32612.tif')
srtm = src_srtm.read(1).astype(float)
srtm[srtm == src_srtm.nodata] = np.nan
fig, ax = plt.subplots()
rasterio.plot.show(src_srtm, cmap='Spectral_r', ax=ax)
fig.colorbar(ax.imshow(srtm, cmap='Spectral_r'), ax=ax);
# Slope
src_srtm_slope = rasterio.open('output/srtm_32612_slope.tif')
srtm_slope = src_srtm_slope.read(1)
srtm_slope[srtm_slope == src_srtm_slope.nodata] = np.nan
fig, ax = plt.subplots()
rasterio.plot.show(src_srtm_slope, cmap='Spectral_r', ax=ax)
fig.colorbar(ax.imshow(srtm_slope, cmap='Spectral_r'), ax=ax);
# Aspect
src_srtm_aspect = rasterio.open('output/srtm_32612_aspect.tif')
srtm_aspect = src_srtm_aspect.read(1)
srtm_aspect[srtm_aspect == src_srtm_aspect.nodata] = np.nan
fig, ax = plt.subplots()
rasterio.plot.show(src_srtm_aspect, cmap='twilight', ax=ax)
fig.colorbar(ax.imshow(srtm_aspect, cmap='twilight'), ax=ax);
```

3.3.5 Zonal operations

Just like focal operations, zonal operations apply an aggregation function to multiple raster cells. However, a second raster, usually with categorical values, defines the zonal filters (or 'zones') in the case of zonal operations, as opposed to a predefined neighborhood window in the case of focal operation presented in the previous section. Consequently, raster cells defining the zonal filter do not necessarily have to be neighbors. Our `grain.tif` raster is a good example, as illustrated in Figure 1.24: different grain sizes are spread irregularly throughout



(a) Input DEM (b) Slope (degrees) (c) Aspect (degrees)

Figure 3.22: Slope and aspect calculation from a DEM

the raster. Finally, the result of a zonal operation is a summary table grouped by zone, which is why this operation is also known as zonal statistics in the GIS world. This is in contrast to focal operations (Section 3.3.4) which return a raster object.

To demonstrate, let's get back to the `grain.tif` and `elev.tif` rasters. To calculate zonal statistics, we use the arrays with raster values, which we already imported earlier. Our intention is to calculate the average (or any other summary function, for that matter) of *elevation* in each zone defined by *grain* values. To do that, first we first obtain the unique values defining the zones using `np.unique`.

```
np.unique(grain)
```

```
array([0, 1, 2], dtype=uint8)
```

Now, we can use dictionary comprehension (see note below) to split the `elev` array into separate one-dimensional arrays with values per `grain` group, with keys being the unique `grain` values.

```
z = {i: elev[grain == i] for i in np.unique(grain)}
z
```

```
{np.uint8(0): array([ 2,  7,  9, 10, 13, 16, 17, 19, 20, 35], dtype=uint8),
 np.uint8(1): array([ 1,  3, 12, 21, 22, 23, 24, 25, 26, 27, 29, 30, 32], dtype=uint8),
 np.uint8(2): array([ 4,  5,  6,  8, 11, 14, 15, 18, 28, 31, 33, 34, 36], dtype=uint8)}
```

> **Note**
>
> *List comprehension* and *dictionary comprehension* are concise ways to create a `list` or a `dict`, respectively, from an iterable object. Both are, conceptually, a concise syntax to replace `for` loops where we iterate over an object and return a same-length object with the results. Here are

minimal examples of list and dictionary comprehension, respectively, to demonstrate the idea:

- `[i**2 for i in [2,4,6]]`—Returns `[4,16,36]`
- `{i: i**2 for i in [2,4,6]}`—Returns `{2:4, 4:16, 6:36}`

List comprehension is more commonly encountered in practice. We use it in Section 4.2.6, Section 5.4.2, Section 5.5.1, and Section 5.6. Dictionary comprehension is only used in one place in the book (this section).

At this stage, we can expand the dictionary comprehension expression to calculate the mean elevation associated with each grain size class. Namely, instead of placing the elevation values (`elev[grain==i]`) into the dictionary values, we place their (rounded) mean (`elev[grain==i].mean().round(1)`).

```
z = {i: elev[grain == i].mean().round(1) for i in np.unique(grain)}
z
```

```
{np.uint8(0): np.float64(14.8),
 np.uint8(1): np.float64(21.2),
 np.uint8(2): np.float64(18.7)}
```

This returns the statistics for each category, here the mean elevation for each grain size class. For example, the mean elevation in pixels characterized by grain size `0` is `14.8`, and so on.

3.3.6 Global operations and distances

Global operations are a special case of zonal operations with the entire raster dataset representing a single zone. The most common global operations are descriptive statistics for the entire raster dataset such as the minimum or maximum—we already discussed those in Section 2.3.2.

Aside from that, global operations are also useful for the computation of distance and weight rasters. In the first case, one can calculate the distance from each cell to specific target cells or vector geometries. For example, one might want to compute the distance to the nearest coast (see Section 5.6). We might also want to consider topography, that means, we are not only interested in the pure distance but would like also to avoid the crossing of mountain ranges when going to the coast. To do so, we can weight the distance with elevation so that each additional altitudinal meter 'prolongs' the Euclidean distance (this is beyond the scope of the book). Visibility and viewshed computations also belong to the family of global operations (also beyond the scope of the book).

3.3.7 Map algebra counterparts in vector processing

Many map algebra operations have a counterpart in vector processing (Liu and Mason 2009). Computing a distance raster (global operation) while only

considering a maximum distance (logical focal operation) is the equivalent of a vector buffer operation (Section 4.2.3). Reclassifying raster data (either local or zonal function depending on the input) is equivalent to dissolving vector data (Section 4.2.7). Overlaying two rasters (local operation), where one contains 'No Data' values representing a mask, is similar to vector clipping (Section 4.2.5). Quite similar to spatial clipping is intersecting two layers (Section 3.2.1, Section 3.2.6). The difference is that these two layers (vector or raster) simply share an overlapping area. However, be careful with the wording. Sometimes the same words have slightly different meanings for raster and vector data models. While aggregating polygon geometries means dissolving boundaries, for raster data geometries it means increasing cell sizes and thereby reducing spatial resolution. Zonal operations dissolve the cells of one raster in accordance with the zones (categories) of another raster dataset using an aggregating function.

3.3.8 Merging rasters

Suppose we would like to compute the NDVI (see Section 3.3.3), and additionally want to compute terrain attributes from elevation data for observations within a study area. Such computations rely on remotely sensed information. The corresponding source imagery is often divided into scenes covering a specific spatial extent (i.e., tiles), and frequently, a study area covers more than one scene. Then, we would need to merge (also known as mosaic) the scenes covering our study area. In case when all scenes are aligned (i.e., share the same origin and resolution), this can be thought of as simply gluing them into one big raster; otherwise, all scenes need to be resampled (see Section 4.3.3) to the same grid (e.g., the one defined by the first scene).

For example, let's merge digital elevation data from two SRTM elevation tiles, for Austria (`'aut.tif'`) and Switzerland (`'ch.tif'`). Merging can be done using function `rasterio.merge.merge`, which accepts a `list` of raster file connections, and returns the new `ndarray` and the corresponding transform object, representing the resulting mosaic.

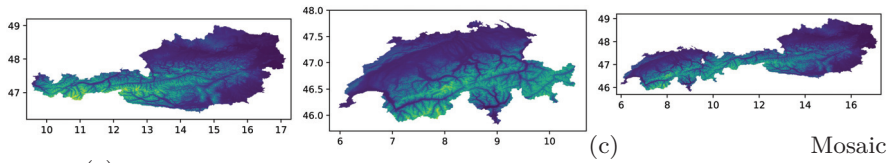
```
src_1 = rasterio.open('data/aut.tif')
src_2 = rasterio.open('data/ch.tif')
out_image, out_transform = rasterio.merge.merge([src_1, src_2])
```

> **Note**
>
> Some Python packages (such as `rasterio`) are split into several so-called sub-modules. The sub-modules are installed collectively when installing the main package. However, each sub-module needs to be loaded separately to be able to use its functions. For example, the `rasterio.merge.merge` function (see last code block) comes from the `rasterio.merge` sub-module of `rasterio`. Loading `rasterio` with

`import rasterio` does not expose the `rasterio.merge.merge` function; instead, we have to load `rasterio.merge` with `import rasterio.merge`, and only then use `rasterio.merge.merge`.
Also check out the first code block in this chapter, where we load `rasterio` as well as three sub-modules: `rasterio.plot`, `rasterio.merge`, and `rasterio.features`.

Figure 3.23 shows both inputs and the resulting mosaic.

```
rasterio.plot.show(src_1);
rasterio.plot.show(src_2);
rasterio.plot.show(out_image, transform=out_transform);
```



(a) `aut.tif` (b) `ch.tif` (c) Mosaic (`aut.tif`+`ch.tif`)

Figure 3.23: Raster merging

By default in `rasterio.merge.merge`, areas of overlap retain the value of the *first* raster (`method='first'`). Other possible methods are:

- `'last'`—Value of the last raster
- `'min'`—Minimum value
- `'max'`—Maximum value

When dealing with non-overlapping tiles, such as `aut.tif` and `ch.tif` (above), the `method` argument has no practical effect. However, it becomes relevant when we want to combine spectral imagery from scenes that were taken on different dates. The above four options for `method` do not cover the commonly required scenario when we would like to compute the *mean* value—for example to calculate a seasonal average NDVI image from a set of partially overlapping satellite images (such as Landsat). An alternative workflow to `rasterio.merge.merge`, for calculating a mosaic as well as averaging any overlaps, is to go through two steps:

- Resampling all scenes into a common 'global' grid (Section 4.3.3), thereby producing a series of matching rasters (with the area surrounding each scene set as 'No Data')
- Averaging the rasters through raster algebra (Section 3.3.3), using `np.mean(m,axis=0)` or `np.nanmean(m,axis=0)` (depending whether we prefer to ignore 'No Data' or not), where `m` is the multi-band array, which would return a single-band array of averages.

4

Geometry operations

Prerequisites

This chapter requires importing the following packages:

```
import sys
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import shapely
import geopandas as gpd
import topojson as tp
import rasterio
import rasterio.plot
import rasterio.warp
import rasterio.mask
```

It also relies on the following data files:

```
seine = gpd.read_file('data/seine.gpkg')
us_states = gpd.read_file('data/us_states.gpkg')
nz = gpd.read_file('data/nz.gpkg')
src = rasterio.open('data/dem.tif')
src_elev = rasterio.open('output/elev.tif')
```

4.1 Introduction

So far the book has explained the structure of geographic datasets (Chapter 1), and how to manipulate them based on their non-geographic attributes (Chapter 2) and spatial relations (Chapter 3). This chapter focuses on manipulating the geographic elements of geographic objects, for example by simplifying and converting vector geometries, and by cropping raster datasets. After reading it you should understand and have control over the geometry column in vector

DOI: 10.1201/9781003379911-4

layers and the extent and geographic location of pixels represented in rasters in relation to other geographic objects.

Section 4.2 covers transforming vector geometries with 'unary' and 'binary' operations. Unary operations work on a single geometry in isolation, including simplification (of lines and polygons), the creation of buffers and centroids, and shifting/scaling/rotating single geometries using 'affine transformations' (Section 4.2.1 to Section 4.2.4). Binary transformations modify one geometry based on the shape of another, including clipping and geometry unions, covered in Section 4.2.5 and Section 4.2.7, respectively. Type transformations (from a polygon to a line, for example) are demonstrated in Section 4.2.8.

Section 4.3 covers geometric transformations on raster objects. This involves changing the size and number of the underlying pixels, and assigning them new values. It teaches how to change the extent and the origin of a raster manually (Section 4.3.1), how to change the resolution in fixed steps through aggregation and disaggregation (Section 4.3.2), and finally how to resample a raster into any existing template, which is the most general and often most practical approach (Section 4.3.3). These operations are especially useful if one would like to align raster datasets from diverse sources. Aligned raster objects share a one-to-one correspondence between pixels, allowing them to be processed using map algebra operations (Section 3.3.3).

In the next chapter (Chapter 5), we deal with the special case of geometry operations that involve both a raster and a vector layer together. It shows how raster values can be 'masked' and 'extracted' by vector geometries. Importantly it shows how to 'polygonize' rasters and 'rasterize' vector datasets, making the two data models more interchangeable.

4.2 Geometric operations on vector data

This section is about operations that in some way change the geometry of vector layers. It is more advanced than the spatial data operations presented in the previous chapter (in Section 3.2), because here we drill down into the geometry: the functions discussed in this section work on the geometric part (the geometry column, which is a `GeoSeries` object), either as standalone object or as part of a `GeoDataFrame`.

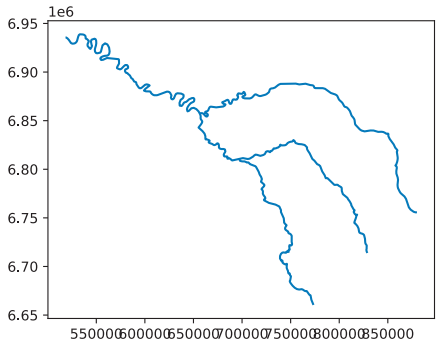
4.2.1 Simplification

Simplification is a process for generalization of vector objects (lines and polygons) usually for use in smaller-scale maps. Another reason for simplifying objects is to reduce the amount of memory, disk space, and network bandwidth

they consume: it may be wise to simplify complex geometries before publishing them as interactive maps. The **geopandas** package provides the `.simplify` method, which uses the GEOS implementation of the Douglas-Peucker algorithm to reduce the vertex count. `.simplify` uses `tolerance` to control the level of generalization in map units (Douglas and Peucker 1973).

For example, a simplified geometry of a `'LineString'` geometry, representing the river Seine and tributaries, using tolerance of 2000 meters, can be created using the `seine.simplify(2000)` command (Figure 4.1).

```
seine_simp = seine.simplify(2000)
seine.plot();
seine_simp.plot();
```



(a) Original (b) Simplified (tolerance = 2000 m)

Figure 4.1: Simplification of the `seine` line layer

The resulting `seine_simp` object is a copy of the original `seine` but with fewer vertices. This is apparent, with the result being visually simpler (Figure 4.1, right) and consuming about twice less memory than the original object, as shown in the comparison below.

```
print(f'Original: {sys.getsizeof(seine)} bytes')
print(f'Simplified: {sys.getsizeof(seine_simp)} bytes')
```

```
Original: 350 bytes
Simplified: 188 bytes
```

Simplification is also applicable for polygons. This is illustrated using `us_states`, representing the contiguous United States. As we show in Chapter 6, for many calculations **geopandas** (through **shapely**, and, ultimately, GEOS) assumes that the data is in a projected CRS and this could lead to unexpected results when applying distance-related operators. Therefore, the first step is to project the data into some adequate projected CRS, such as

US National Atlas Equal Area (EPSG:9311) (on the left in Figure 4.2), using `.to_crs` (Section 6.7).

```
us_states9311 = us_states.to_crs(9311)
```

The `.simplify` method from **geopandas** works the same way with a `'Polygon'/'MultiPolygon'` layer such as `us_states9311`:

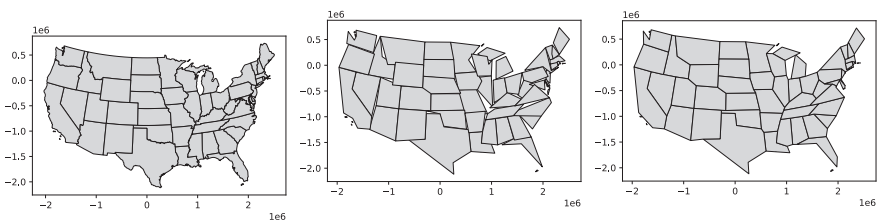
```
us_states_simp1 = us_states9311.simplify(100000)
```

A limitation with `.simplify`, however, is that it simplifies objects on a per-geometry basis. This means the topology is lost, resulting in overlapping and 'holey' areal units as illustrated in Figure 4.2 (b). The `.toposimplify` method from package **topojson** provides an alternative that overcomes this issue. The main advantage of `.toposimplify` is that it is topologically 'aware': it simplifies the combined borders of the polygons (rather than each polygon on its own), thus ensuring that the overlap is maintained. The following code chunk uses `.toposimplify` to simplify `us_states9311`. Note that, when using the **topojson** package, we first need to calculate a topology object, using function `tp.Topology`, and then apply the simplification function, such as `.toposimplify`, to obtain a simplified layer. We are also using the `.to_gdf` method to return a `GeoDataFrame`.

```
topo = tp.Topology(us_states9311, prequantize=False)
us_states_simp2 = topo.toposimplify(100000).to_gdf()
```

Figure 4.2 compares the original input polygons and two simplification methods applied to `us_states9311`.

```
us_states9311.plot(color='lightgrey', edgecolor='black');
us_states_simp1.plot(color='lightgrey', edgecolor='black');
us_states_simp2.plot(color='lightgrey', edgecolor='black');
```



(a) Original

(b) Simplified using **geopandas**

(c) Simplified using **topojson**

Figure 4.2: Polygon simplification in action, comparing the original geometry of the contiguous United States with simplified versions, generated with functions from the **geopandas** (middle), and **topojson** (right), packages.

4.2.2 Centroids

Centroid operations identify the center of geographic objects. Like statistical measures of central tendency (including mean and median definitions of 'average'), there are many ways to define the geographic center of an object. All of them create single-point representations of more complex vector objects.

The most commonly used centroid operation is the geographic centroid. This type of centroid operation (often referred to as 'the centroid') represents the center of mass in a spatial object (think of balancing a plate on your finger). Geographic centroids have many uses, for example to create a simple point representation of complex geometries, to estimate distances between polygons, or to specify the location where polygon text labels are placed. Centroids of the geometries in a `GeoSeries` or a `GeoDataFrame` are accessible through the `.centroid` property, as demonstrated in the code below, which generates the geographic centroids of regions in New Zealand and tributaries to the River Seine (black points in Figure 4.3).

```
nz_centroid = nz.centroid
seine_centroid = seine.centroid
```

Sometimes the geographic centroid falls outside the boundaries of their parent objects (think of vector data in shape of a doughnut). In such cases 'point on surface' operations, created with the `.representative_point` method, can be used to guarantee the point will be in the parent object (e.g., for labeling irregular multipolygon objects such as island states), as illustrated by the red points in Figure 4.3. Notice that these red points always lie on their parent objects.

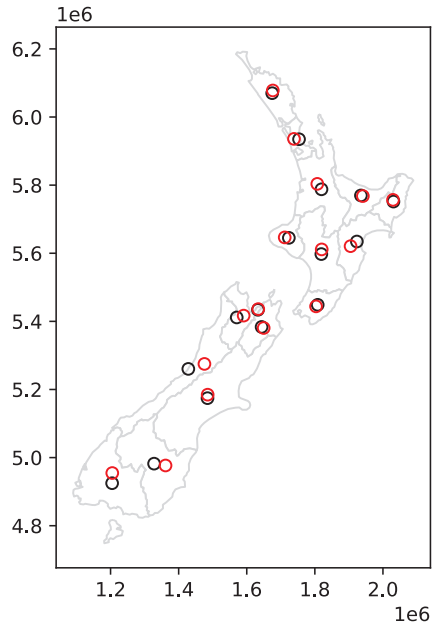
```
nz_pos = nz.representative_point()
seine_pos = seine.representative_point()
```

The centroids and points on surface are illustrated in Figure 4.3.

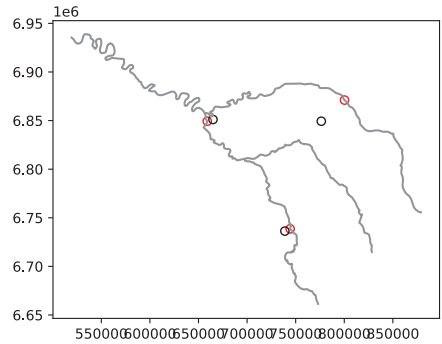
```
# New Zealand
base = nz.plot(color='white', edgecolor='lightgrey')
nz_centroid.plot(ax=base, color='None', edgecolor='black')
nz_pos.plot(ax=base, color='None', edgecolor='red');
# Seine
base = seine.plot(color='grey')
seine_pos.plot(ax=base, color='None', edgecolor='red')
seine_centroid.plot(ax=base, color='None', edgecolor='black');
```

4.2.3 Buffers

Buffers are polygons representing the area within a given distance of a geometric feature: regardless of whether the input is a point, line or polygon, the output is a polygon (when using positive buffer distance). Unlike simplification, which



(a) New Zealand



(b) Seine

Figure 4.3: Centroids (black) and points on surface (red) of New Zealand and Seine datasets.

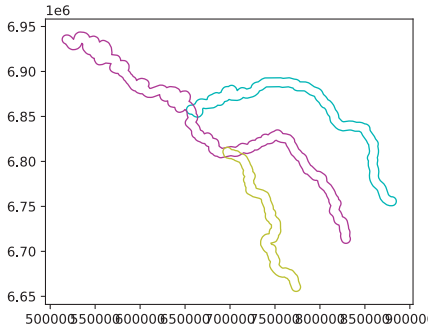
is often used for visualization and reducing file size, buffering tends to be used for geographic data analysis. How many points are within a given distance of this line? Which demographic groups are within travel distance of this new shop? These kinds of questions can be answered and visualized by creating buffers around the geographic entities of interest.

Figure 4.4 illustrates buffers of two different sizes (5 and 50 km) surrounding the river Seine and tributaries. These buffers were created with commands below, using the `.buffer` method, applied to a `GeoSeries` or `GeoDataFrame`. The `.buffer` method requires one important argument: the buffer distance, provided in the units of the CRS, in this case, meters.

```
seine_buff_5km = seine.buffer(5000)
seine_buff_50km = seine.buffer(50000)
```

The results are shown in Figure 4.4.

```
seine_buff_5km.plot(color='none', edgecolor=['c', 'm', 'y']);
seine_buff_50km.plot(color='none', edgecolor=['c', 'm', 'y']);
```



(a) 5 *km* buffer

(b) 50 *km* buffer

Figure 4.4: Buffers around the Seine dataset of 5 *km* and 50 *km*. Note the colors, which reflect the fact that one buffer is created per geometry feature.

Note that both `.centroid` and `.buffer` return a `GeoSeries` object, even when the input is a `GeoDataFrame`.

```
seine_buff_5km
```

```
0    POLYGON ((657550.332 6852587.97...
1    POLYGON ((517151.801 6930724.10...
2    POLYGON ((701519.74 6813075.492...
dtype: geometry
```

In the common scenario when the original attributes of the input features need to be retained, you can replace the existing geometry with the new `GeoSeries` by creating a copy of the original `GeoDataFrame` and assigning the new buffer `GeoSeries` to the `geometry` column.

```
seine_buff_5km = seine.copy()
seine_buff_5km.geometry = seine.buffer(5000)
seine_buff_5km
```

	name	geometry
0	Marne	POLYGON ((657550.332 6852587.97...
1	Seine	POLYGON ((517151.801 6930724.10...
2	Yonne	POLYGON ((701519.74 6813075.492...

An alternative option is to add a secondary geometry column directly to the original `GeoDataFrame`.

```
seine['geometry_5km'] = seine.buffer(5000)
seine
```

	name	geometry	geometry_5km
0	Marne	MULTILINESTRING ((879955.277 67...	POLYGON ((657550.332 6852587.97...
1	Seine	MULTILINESTRING ((828893.615 67...	POLYGON ((517151.801 6930724.10...
2	Yonne	MULTILINESTRING ((773482.137 66...	POLYGON ((701519.74 6813075.492...

You can then switch to either geometry column (i.e., make it 'active') using `.set_geometry`, as in:

```
seine = seine.set_geometry('geometry_5km')
```

Let's revert to the original state of `seine` before moving on to the next section.

```
seine = seine.set_geometry('geometry')
seine = seine.drop('geometry_5km', axis=1)
```

4.2.4 Affine transformations

Affine transformations include, among others, shifting (translation), scaling and rotation, or any combination of these. They preserve lines and parallelism, but angles and lengths are not necessarily preserved. These transformations are an essential part of geocomputation. For example, shifting is needed for labels placement, scaling is used in non-contiguous area cartograms, and many affine transformations are applied when reprojecting or improving the geometry that was created based on a distorted or wrongly projected map.

The **geopandas** package implements affine transformation, for objects of classes `GeoSeries` and `GeoDataFrame`. In both cases, the method is applied on the `GeoSeries` part, returning just the `GeoSeries` of transformed geometries.

Affine transformations of `GeoSeries` can be done using the `.affine_transform` method, which is a wrapper around the `shapely.affinity.affine_transform` function. A two-dimensional affine transformation requires a six-parameter list `[a,b,d,e,xoff,yoff]` which represents Equation 4.1 and Equation 4.2 for transforming the coordinates.

$$x' = ax + by + x_{\text{off}} \tag{4.1}$$

$$y' = dx + ey + y_{\text{off}} \tag{4.2}$$

There are also simplified `GeoSeries` methods for specific scenarios, such as:

- `.translate(xoff=0.0, yoff=0.0)`
- `.scale(xfact=1.0, yfact=1.0, origin='center')`
- `.rotate(angle, origin='center', use_radians=False)`

For example, *shifting* only requires the x_{off} and y_{off}, using `.translate`. The code below shifts the y-coordinates of `nz` by 100 km to the north, but leaves the x-coordinates untouched.

```
nz_shift = nz.translate(0, 100000)
nz_shift
```

```
0     MULTIPOLYGON (((1745493.196 610...
1     MULTIPOLYGON (((1803822.103 600...
2     MULTIPOLYGON (((1860345.005 595...
                       ...
13    MULTIPOLYGON (((1616642.877 552...
14    MULTIPOLYGON (((1624866.278 551...
15    MULTIPOLYGON (((1686901.914 545...
Length: 16, dtype: geometry
```

> **Note**
>
> **shapely**, and consequently **geopandas**, operations, typically ignore the z-dimension (if there is one) of geometries in operations. For example, `shapely.LineString([(0,0,0),(0,0,1)]).length` returns `0` (and not `1`), since `.length` ignores the z-dimension. This is not an issue in this book (and in most real-world spatial analysis applications), since we are dealing only with two-dimensional geometries.

Scaling enlarges or shrinks objects by a factor, and can be applied either globally or locally. Global scaling increases or decreases all coordinates values in relation to the origin coordinates, while keeping all geometries topological relations intact. **geopandas** implements scaling using the `.scale` method. Local scaling treats geometries independently and requires points around which geometries are going to be scaled, e.g., centroids. In the example below, each geometry is shrunk by a factor of two around the centroids (Figure 4.5 (b)). To achieve that, we pass the `0.5` and `0.5` scaling factors (for x and y, respectively), and the `'centroid'` option for the point of origin.

```
nz_scale = nz.scale(0.5, 0.5, origin='centroid')
nz_scale
```

```
0     MULTIPOLYGON (((1710099.077 603...
1     MULTIPOLYGON (((1778686.524 591...
2     MULTIPOLYGON (((1839927.904 582...
                       ...
13    MULTIPOLYGON (((1593619.59 5418...
14    MULTIPOLYGON (((1628907.395 542...
15    MULTIPOLYGON (((1665262.436 536...
Length: 16, dtype: geometry
```

When setting the `origin` in `.scale`, other than `'centroid'` it is possible to use `'center'`, for the bounding box center, or specific point coordinates, such as `(0,0)`.

Rotating the geometries can be done using the `.rotate` method. When rotating, we need to specify the rotation angle (positive values imply clockwise rotation) and the `origin` points (using the same options as in `.scale`). For example, the following expression rotates `nz` by 30° counter-clockwise, around the geometry centroids.

```
nz_rotate = nz.rotate(-30, origin='centroid')
nz_rotate
```

```
0     MULTIPOLYGON (((1701904.887 597...
1     MULTIPOLYGON (((1779714.772 587...
2     MULTIPOLYGON (((1890843.462 582...
                        ...
13    MULTIPOLYGON (((1616991.636 539...
14    MULTIPOLYGON (((1617733.547 542...
15    MULTIPOLYGON (((1665898.669 533...
Length: 16, dtype: geometry
```

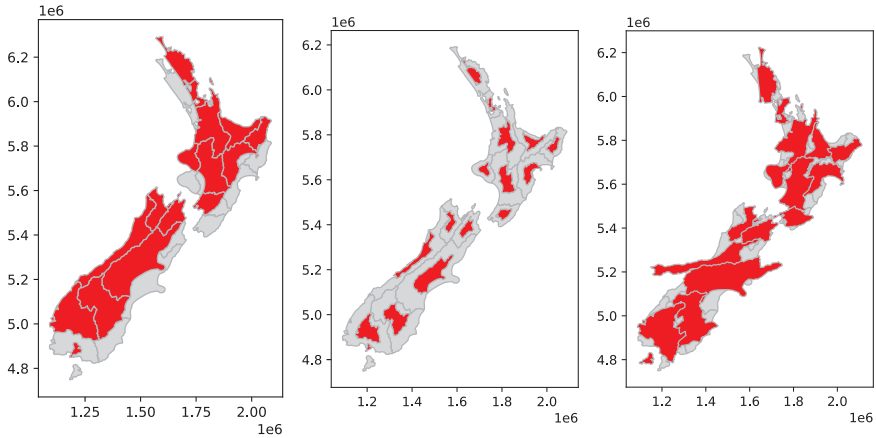
Figure 4.5 shows the original layer `nz`, and the shifting, scaling, and rotation results.

```
# Shift
base = nz.plot(color='lightgrey', edgecolor='darkgrey')
nz_shift.plot(ax=base, color='red', edgecolor='darkgrey');
# Scale
base = nz.plot(color='lightgrey', edgecolor='darkgrey')
nz_scale.plot(ax=base, color='red', edgecolor='darkgrey');
# Rotate
base = nz.plot(color='lightgrey', edgecolor='darkgrey')
nz_rotate.plot(ax=base, color='red', edgecolor='darkgrey');
```

4.2.5 Pairwise geometry-generating operations

Spatial clipping is a form of spatial subsetting that involves changes to the geometry columns of at least some of the affected features. Clipping can only apply to features more complex than points: lines, polygons, and their 'multi' equivalents. To illustrate this concept, we will start with a simple example: two overlapping circles with a center point one unit away from each other and a radius of one (Figure 4.6).

```
x = shapely.Point((0, 0)).buffer(1)
y = shapely.Point((1, 0)).buffer(1)
shapely.GeometryCollection([x, y])
```



(a) Shift (b) Scale (c) Rotate

Figure 4.5: Affine transformations of the `nz` layer: shift, scale, and rotate

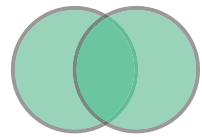


Figure 4.6: Overlapping polygon (circle) geometries `x` and `y`

Imagine you want to select not one circle or the other, but the space covered by both `x` and `y`. This can be done using the `.intersection` method from **shapely**, illustrated using objects named `x` and `y` which represent the left- and right-hand circles (Figure 4.7).

```
x.intersection(y)
```

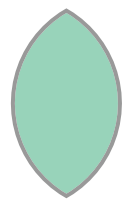


Figure 4.7: Intersection between `x` and `y`

More generally, clipping is an example of a 'pairwise geometry-generating operation', where new geometries are generated from two inputs. Other than `.intersection` (Figure 4.7), there are three other standard pairwise operators: `.difference` (Figure 4.8), `.union` (Figure 4.9), and `.symmetric_difference` (Figure 4.10).

```
x.difference(y)
```

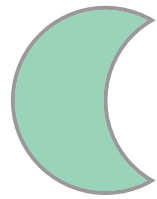


Figure 4.8: Difference between `x` and `y` (namely, `x` 'minus' `y`)

```
x.union(y)
```

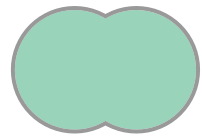


Figure 4.9: Union of `x` and `y`

```
x.symmetric_difference(y)
```

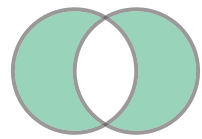


Figure 4.10: Symmetric difference between `x` and `y`

Keep in mind that `x` and `y` are interchangeable in all predicates except for `.difference`, where `x.difference(y)` means `x` minus `y`, whereas `y.difference(x)` means `y` minus `x`.

The latter examples demonstrate pairwise operations between individual `shapely` geometries. The **geopandas** package, as is often the case, contains wrappers of these **shapely** functions to be applied to multiple, or pairwise, use cases. For example, applying either of the pairwise methods on a `GeoSeries` or `GeoDataFrame`, combined with a `shapely` geometry, returns the pairwise (many-to-one) results (which is analogous to other operators, like `.intersects` or `.distance`, see Section 3.2.1 and Section 3.2.7, respectively).

Let's demonstrate the 'many-to-one' scenario by calculating the difference between each geometry in a `GeoSeries` and a fixed `shapely` geometry. To create the latter, let's take `x` and combine it with itself translated (Section 4.2.4) to a distance of `1` and `2` units 'upwards' on the y-axis.

```
geom1 = gpd.GeoSeries(x)
geom2 = geom1.translate(0, 1)
geom3 = geom1.translate(0, 2)
geom = pd.concat([geom1, geom2, geom3])
geom
```

```
0    POLYGON ((1 0, 0.99518 -0.09802...
0    POLYGON ((1 1, 0.99518 0.90198,...
0    POLYGON ((1 2, 0.99518 1.90198,...
dtype: geometry
```

Figure 4.11 shows the `GeoSeries` `geom` with the `shapely` geometry (in red) that we will intersect with it.

```
fig, ax = plt.subplots()
geom.plot(color='#00000030', edgecolor='black', ax=ax)
gpd.GeoSeries(y).plot(color='#FF000040', edgecolor='black', ax=ax);
```

Now, using `.intersection` automatically applies the **shapely** method of the same name on each geometry in `geom`, returning a new `GeoSeries`, which we name `geom_inter_y`, with the pairwise intersections. Note the empty third geometry (can you explain the meaning of this result?).

```
geom_inter_y = geom.intersection(y)
geom_inter_y
```

```
0    POLYGON ((0.99518 -0.09802, 0.9...
0    POLYGON ((0.99518 0.90198, 0.98...
0                        POLYGON EMPTY
dtype: geometry
```

Figure 4.12 is a plot of the result `geom_inter_y`.

```
geom_inter_y.plot(color='#00000030', edgecolor='black');
```

The `.overlay` method (see Section 3.2.6) further extends this technique, making it possible to apply 'many-to-many' pairwise geometry generations between

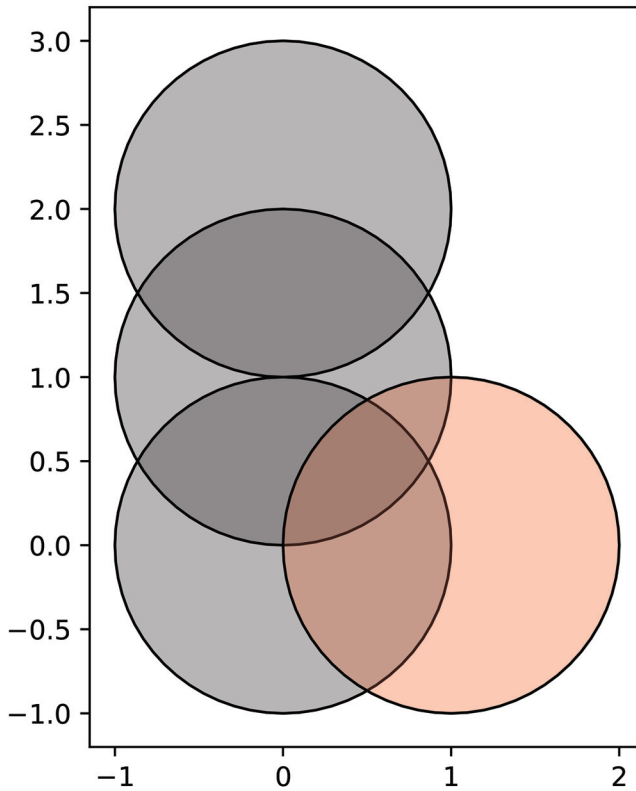


Figure 4.11: A `GeoSeries` with three circles (in grey), and a `shapely` geometry that we will subtract from it (in red)

all pairs of two `GeoDataFrame`s. The output is a new `GeoDataFrame` with the pairwise outputs, plus the attributes of both inputs which were the inputs of the particular pairwise output geometry. Also see the *Set operations with overlay*[1] article in the **geopandas** documentation for examples of `.overlay`.

4.2.6 Subsetting vs. clipping

In the last two chapters we have introduced two types of spatial operators: boolean, such as `.intersects` (Section 3.2.1), and geometry-generating, such as `.intersection` (Section 4.2.5). Here, we illustrate the difference between them. We do this using the specific scenario of subsetting points by polygons, where (unlike in other cases) both methods can be used for the same purpose and giving the same result.

[1] https://geopandas.org/en/stable/docs/user_guide/set_operations.html

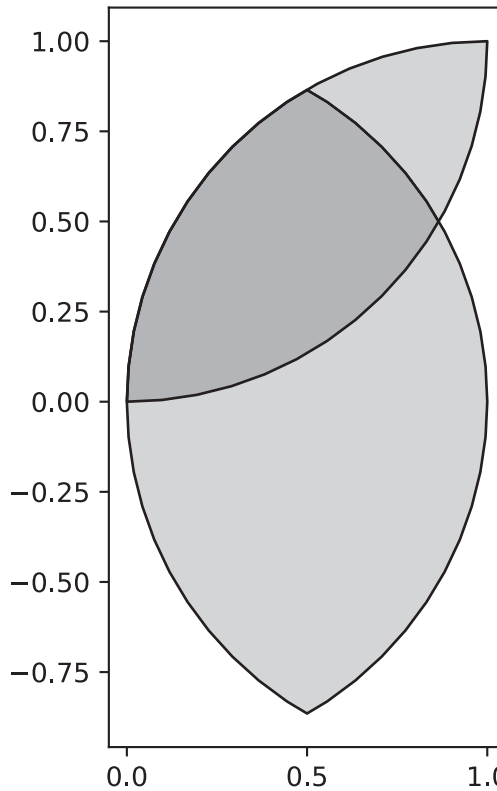


Figure 4.12: The output `GeoSeries`, after subtracting a `shapely` geometry using `.intersection`

To illustrate the point, we will subset points that cover the bounding box of the circles `x` and `y` from Figure 4.6. Some points will be inside just one circle, some will be inside both, and some will be inside neither. The following code sections generate the sample data for this section, a simple random distribution of points within the extent of circles `x` and `y`, resulting in output illustrated in Figure 4.13. We create the sample points in two steps. First, we figure out the bounds where random points are to be generated.

```
bounds = x.union(y).bounds
bounds
```

```
(-1.0, -1.0, 2.0, 1.0)
```

Second, we use `np.random.uniform` to calculate `n` random x- and y-coordinates within the given bounds.

```
np.random.seed(1)
n = 10
coords_x = np.random.uniform(bounds[0], bounds[2], n)
coords_y = np.random.uniform(bounds[1], bounds[3], n)
coords = list(zip(coords_x, coords_y))
coords
```

```
[(np.float64(0.2510660141077219), np.float64(-0.1616109711934104)),
 (np.float64(1.1609734803264744), np.float64(0.370439000793519)),
 (np.float64(-0.9996568755479653), np.float64(-0.5910955005369651)),
 (np.float64(-0.0930022821044807), np.float64(0.7562348727818908)),
 (np.float64(-0.5597323275486609), np.float64(-0.9452248136041477)),
 (np.float64(-0.7229842156936066), np.float64(0.34093502035680445)),
 (np.float64(-0.4412193658669873), np.float64(-0.16539039526574606)),
 (np.float64(0.03668218112914312), np.float64(0.11737965689150331)),
 (np.float64(0.1903024226920098), np.float64(-0.7192261228095325)),
 (np.float64(0.6164502020100708), np.float64(-0.6037970218302424))]
```

Third, we transform the list of coordinates into a `list` of `shapely` points, and then to a `GeoSeries`.

```
pnt = [shapely.Point(i) for i in coords]
pnt = gpd.GeoSeries(pnt)
```

The result `pnt`, with `x` and `y` circles in the background, is shown in Figure 4.13.

```
base = pnt.plot(color='none', edgecolor='black')
gpd.GeoSeries(x).plot(ax=base, color='none', edgecolor='darkgrey');
gpd.GeoSeries(y).plot(ax=base, color='none', edgecolor='darkgrey');
```

Now, we can get back to our question: how to subset the points to only return the points that intersect with both `x` and `y`? The code chunks below demonstrate two ways to achieve the same result. In the first approach, we can calculate a boolean `Series`, evaluating whether each point of `pnt` intersects with the intersection of `x` and `y` (see Section 3.2.1), and then use it to subset `pnt` to get the result `pnt1`.

```
sel = pnt.intersects(x.intersection(y))
pnt1 = pnt[sel]
pnt1
```

```
0    POINT (0.25107 -0.16161)
7     POINT (0.03668 0.11738)
9     POINT (0.61645 -0.6038)
dtype: geometry
```

In the second approach, we can also find the intersection between the input points represented by `pnt`, using the intersection of `x` and `y` as the subsetting/clipping object. Since the second argument is an individual `shapely`

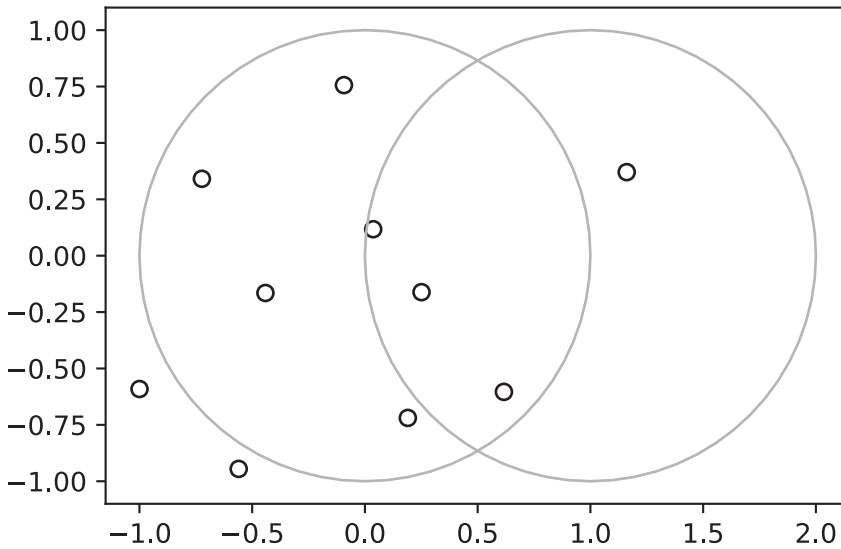


Figure 4.13: Randomly distributed points within the bounding box enclosing circles `x` and `y`

geometry (`x.intersection(y)`), we get 'pairwise' intersections of each `pnt` with it (see Section 4.2.5):

```
pnt2 = pnt.intersection(x.intersection(y))
pnt2
```

```
0    POINT (0.25107 -0.16161)
1                POINT EMPTY
2                POINT EMPTY
                 ...
7     POINT (0.03668 0.11738)
8                POINT EMPTY
9     POINT (0.61645 -0.6038)
Length: 10, dtype: geometry
```

The subset `pnt2` is shown in Figure 4.14.

```
base = pnt.plot(color='none', edgecolor='black')
gpd.GeoSeries(x).plot(ax=base, color='none', edgecolor='darkgrey');
gpd.GeoSeries(y).plot(ax=base, color='none', edgecolor='darkgrey');
pnt2.plot(ax=base, color='red');
```

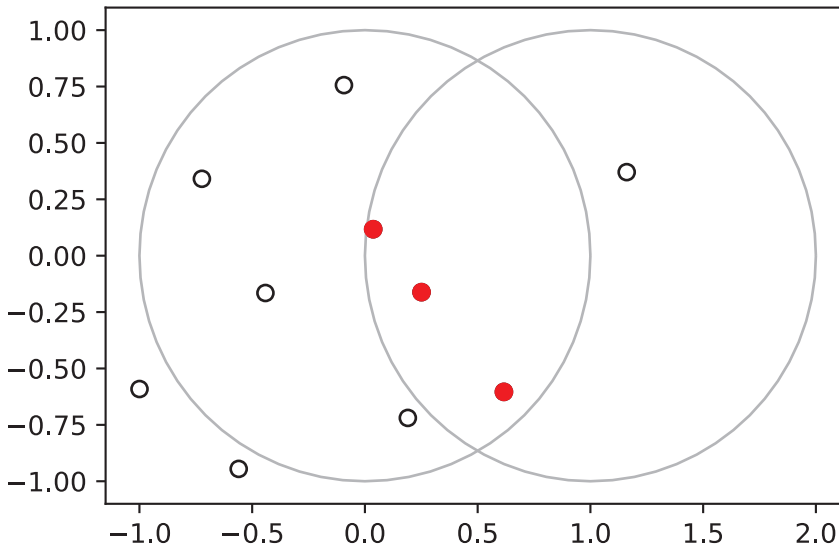


Figure 4.14: Randomly distributed points within the bounding box enclosing circles `x` and `y`. The points that intersect with both objects `x` and `y` are highlighted.

The only difference between the two approaches is that `.intersection` returns all intersections, even if they are empty. When these are filtered out, `pnt2` becomes identical to `pnt1`:

```
pnt2 = pnt2[~pnt2.is_empty]
pnt2
```

```
0    POINT (0.25107 -0.16161)
7     POINT (0.03668 0.11738)
9     POINT (0.61645 -0.6038)
dtype: geometry
```

The example above is rather contrived and provided for educational rather than applied purposes. However, we encourage the reader to reproduce the results to deepen your understanding of handling geographic vector objects in Python.

4.2.7 Geometry unions

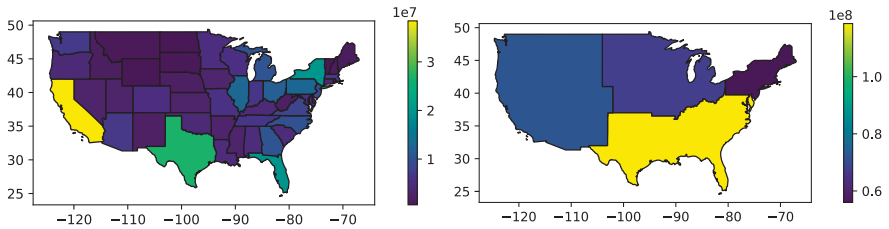
Spatial aggregation can silently dissolve the geometries of touching polygons in the same group, as we saw in Section 2.2.2. This is demonstrated in the code chunk below, in which 49 `us_states` are aggregated into 4 regions using the `.dissolve` method.

```
regions = us_states[['REGION', 'geometry', 'total_pop_15']] \
    .dissolve(by='REGION', aggfunc='sum').reset_index()
regions
```

	REGION	geometry	total_pop_15
0	Midwest	MULTIPOLYGON (((-89.10077 36.94...	67546398.0
1	Northeast	MULTIPOLYGON (((-75.61724 39.83...	55989520.0
2	South	MULTIPOLYGON (((-81.3855 30.273...	118575377.0
3	West	MULTIPOLYGON (((-118.36998 32.8...	72264052.0

Figure 4.15 compares the original `us_states` layer with the aggregated `regions` layer.

```
# States
fig, ax = plt.subplots(figsize=(9, 2.5))
us_states.plot(ax=ax, edgecolor='black', column='total_pop_15', legend=True);
# Regions
fig, ax = plt.subplots(figsize=(9, 2.5))
regions.plot(ax=ax, edgecolor='black', column='total_pop_15', legend=True);
```



(a) 49 States (b) 4 Regions

Figure 4.15: Spatial aggregation on contiguous polygons, illustrated by aggregating the population of 49 US states into 4 regions, with population represented by color. Note the operation automatically dissolves boundaries between states.

What is happening with the geometries here? Behind the scenes, `.dissolve` combines the geometries and dissolves the boundaries between them using the `.union_all` method per group. This is demonstrated in the code chunk below which creates a united western US using the standalone `.union_all` operation. Note that the result is a `shapely` geometry, as the individual attributes are 'lost' as part of dissolving (Figure 4.16).

```
us_west = us_states[us_states['REGION'] == 'West']
us_west_union = us_west.geometry.union_all()
us_west_union
```

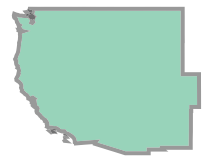


Figure 4.16: Western US

To dissolve two (or more) groups of a `GeoDataFrame` into one geometry, we can either (a) use a combined condition or (b) concatenate the two separate subsets and then dissolve using `.union_all`.

```
# Approach 1
sel = (us_states['REGION'] == 'West') | (us_states['NAME'] == 'Texas')
texas_union = us_states[sel]
texas_union = texas_union.geometry.union_all()
# Approach 2
us_west = us_states[us_states['REGION'] == 'West']
texas = us_states[us_states['NAME'] == 'Texas']
texas_union = pd.concat([us_west, texas]).union_all()
```

The result is identical in both cases, shown in Figure 4.17.

```
texas_union
```

Figure 4.17: Western US and Texas

4.2.8 Type transformations

Transformation of geometries, from one type to another, also known as 'geometry casting', is often required to facilitate spatial analysis. Either the **geopandas** or the **shapely** packages can be used for geometry casting, depending on the type of transformation, and the way that the input is organized (whether as individual geometry, or a vector layer). Therefore, the exact expression(s) depend on the specific transformation we are interested in.

In general, you need to figure out the required input of the respective constructor function according to the 'destination' geometry (e.g., `shapely.LineString`, etc.), then reshape the input of the source geometry

into the right form to be passed to that function. Or, when available, you can use a wrapper from **geopandas**.

In this section, we demonstrate several common scenarios. We start with transformations of individual geometries from one type to another, using **shapely** methods:

- `'MultiPoint'` to `'LineString'` (Figure 4.19)
- `'MultiPoint'` to `'Polygon'` (Figure 4.20)
- `'LineString'` to `'MultiPoint'` (Figure 4.22)
- `'Polygon'` to `'MultiPoint'` (Figure 4.23)
- `'Polygon'`s to `'MultiPolygon'` (Figure 4.24)
- `'MultiPolygon'`s to `'Polygon'`s (Figure 4.25, Figure 4.26)

Then, we move on and demonstrate casting workflows on `GeoDataFrame`s, where we have further considerations, such as keeping track of geometry attributes, and the possibility of dissolving, rather than just combining, geometries. As we will see, these are done either by manually applying **shapely** methods on all geometries in the given layer, or using **geopandas** wrapper methods which do it automatically:

- `'MultiLineString'` to `'LineString'`s (using `.explode`) (Figure 4.28)
- `'LineString'` to `'MultiPoint'`s (using `.apply`) (Figure 4.29)
- `'LineString'`s to `'MultiLineString'` (using `.dissolve`)
- `'Polygon'`s to `'MultiPolygon'` (using `.dissolve` or `.agg`) (Figure 4.30)
- `'Polygon'` to `'(Multi)LineString'` (using `.boundary` or `.exterior`) (demonstrated in a subsequent chapter, see Section 5.4.2)

Let's start with the simple individual-geometry casting examples, to illustrate how geometry casting works on **shapely** geometry objects. First, let's create a `'MultiPoint'` (Figure 4.18).

```
multipoint = shapely.MultiPoint([(1,1), (3,3), (5,1)])
multipoint
```

Figure 4.18: A `'MultiPoint'` geometry used to demonstrate **shapely** type transformations

A `'LineString'` can be created using `shapely.LineString` from a `list` of points. Thus, a `'MultiPoint'` can be converted to a `'LineString'` by passing the points into a `list`, then passing them to `shapely.LineString` (Figure 4.19). The `.geoms` property, mentioned in Section 1.2.5, gives access to the individual parts that comprise a multi-part geometry, as an iterable

object similar to a `list`; it is one of the **shapely** access methods to internal parts of a geometry.

```
linestring = shapely.LineString(multipoint.geoms)
linestring
```

Figure 4.19: A `'LineString'` created from the `'MultiPoint'` in Figure 4.18

Similarly, a `'Polygon'` can be created using function `shapely.Polygon`, which accepts a sequence of point coordinates. In principle, the last coordinate must be equal to the first, in order to form a closed shape. However, `shapely.Polygon` is able to complete the last coordinate automatically, and therefore we can pass all of the coordinates of the `'MultiPoint'` directly to `shapely.Polygon` (Figure 4.20).

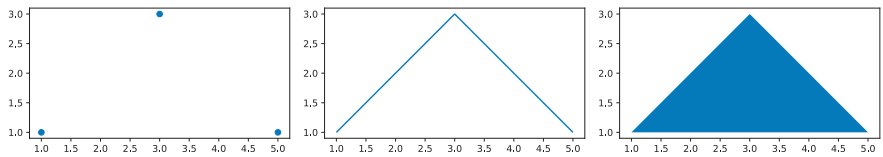
```
polygon = shapely.Polygon(multipoint.geoms)
polygon
```

Figure 4.20: A `'Polygon'` created from the `'MultiPoint'` in Figure 4.18

The source `'MultiPoint'` geometry, and the derived `'LineString'` and `'Polygon'` geometries are shown in Figure 4.21. Note that we convert the `shapely` geometries to `GeoSeries` to be able to use the **geopandas** `.plot` method.

```
gpd.GeoSeries(multipoint).plot();
gpd.GeoSeries(linestring).plot();
gpd.GeoSeries(polygon).plot();
```

Conversion from `'MultiPoint'` to `'LineString'`, shown above (Figure 4.19), is a common operation that creates a line object from ordered point observations, such as GPS measurements or geotagged media. This allows spatial operations, such as calculating the length of the path traveled. Conversion from `'MultiPoint'` or `'LineString'` to `'Polygon'` (Figure 4.20) is often used to calculate an area, for example from the set of GPS measurements taken around a lake or from the corners of a building lot.



(a) `'MultiPoint'` (b) `'LineString'` (c) `'Polygon'`

Figure 4.21: Examples of `'LineString'` and `'Polygon'` casted from a `'MultiPoint'` geometry

Our `'LineString'` geometry can be converted back to a `'MultiPoint'` geometry by passing its coordinates directly to `shapely.MultiPoint` (Figure 4.22).

```
shapely.MultiPoint(linestring.coords)
```

Figure 4.22: A `'MultiPoint'` created from the `'LineString'` in Figure 4.19

A `'Polygon'` (exterior) coordinates can be passed to `shapely.MultiPoint`, to go back to a `'MultiPoint'` geometry, as well (Figure 4.23).

```
shapely.MultiPoint(polygon.exterior.coords)
```

Figure 4.23: A `'MultiPoint'` created from the `'Polygon'` in Figure 4.20

Using these methods, we can transform between `'Point'`, `'LineString'`, and `'Polygon'` geometries, assuming there is a sufficient number of points (at least two for a line, and at least three for a polygon). When dealing with multi-part geometries using **shapely**, we can:

- Access single-part geometries (e.g., each `'Polygon'` in a `'MultiPolygon'` geometry) using `.geoms[i]`, where `i` is the index of the geometry
- Combine single-part geometries into a multi-part geometry, by passing a `list` of the latter to the constructor function

For example, here is how we combine two `'Polygon'` geometries into a `'MultiPolygon'` (while also using a **shapely** affine function `shapely.affinity.translate`, which is underlying the **geopandas** `.translate` method used earlier, see Section 4.2.4) (Figure 4.24):

```
multipolygon = shapely.MultiPolygon([
    polygon,
    shapely.affinity.translate(polygon.centroid.buffer(1.5), 3, 2)
])
multipolygon
```

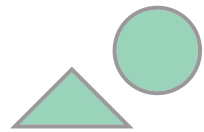


Figure 4.24: A `'MultiPolygon'` created from the `'Polygon'` in Figure 4.20 and another polygon

Given `multipolygon`, here is how we can get back the `'Polygon'` part 1 (Figure 4.25):

```
multipolygon.geoms[0]
```

Figure 4.25: The 1st part extracted from the `'MultiPolygon'` in Figure 4.24

and part 2 (Figure 4.26):

```
multipolygon.geoms[1]
```

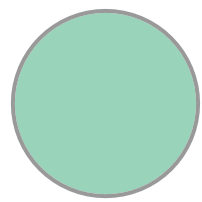


Figure 4.26: The 2nd part extracted from the `'MultiPolygon'` in Figure 4.24

However, dealing with multi-part geometries can be easier with **geopandas**. Thanks to the fact that geometries in a `GeoDataFrame` are associated with attributes, we can keep track of the origin of each geometry: duplicating the attributes when going from multi-part to single-part (using `.explode`, see below), or 'collapsing' the attributes through aggregation when going from single-part to multi-part (using `.dissolve`, see Section 4.2.7).

Let's demonstrate going from multi-part to single-part (Figure 4.28) and then back to multi-part (Section 4.2.7), using a small line layer. As input, we will create a `'MultiLineString'` geometry composed of three lines (Figure 4.27).

```
l1 = shapely.LineString([(1, 5), (4, 3)])
l2 = shapely.LineString([(4, 4), (4, 1)])
l3 = shapely.LineString([(2, 2), (4, 2)])
ml = shapely.MultiLineString([l1, l2, l3])
ml
```

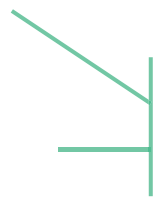


Figure 4.27: A `'MultiLineString'` geometry composed of three lines

Let's place it into a `GeoSeries`.

```
geom = gpd.GeoSeries(ml)
geom
```

```
0    MULTILINESTRING ((1 5, 4 3), (4...
dtype: geometry
```

Then, put it into a `GeoDataFrame` with an attribute called `'id'`:

```
dat = gpd.GeoDataFrame(geometry=geom, data=pd.DataFrame({'id': [1]}))
dat
```

	id	geometry
0	1	MULTILINESTRING ((1 5, 4 3), (4...

You can imagine it as a road or river network. The above layer `dat` has only one row that defines all the lines. This restricts the number of operations that can be done, for example, it prevents adding names to each line segment or calculating lengths of single lines. Using **shapely** methods with which we are

already familiar with (see above), the individual single-part geometries (i.e., the 'parts') can be accessed through the `.geoms` property.

```
list(ml.geoms)
```

```
[<LINESTRING (1 5, 4 3)>, <LINESTRING (4 4, 4 1)>, <LINESTRING (2 2, 4 2)>]
```

However, specifically for the 'multi-part to single part' type transformation scenario, there is also a method called `.explode`, which can convert an entire multi-part `GeoDataFrame` to a single-part one. The advantage is that the original attributes (such as `id`) are retained, so that we can keep track of the original multi-part geometry properties that each part came from. The `index_parts=True` argument also lets us keep track of the original multipart geometry indices, and part indices, named `level_0` and `level_1`, respectively.

```
dat1 = dat.explode(index_parts=True).reset_index()
dat1
```

	level_0	level_1	id	geometry
0	0	0	1	LINESTRING (1 5, 4 3)
1	0	1	1	LINESTRING (4 4, 4 1)
2	0	2	1	LINESTRING (2 2, 4 2)

For example, here we see that all `'LineString'` geometries came from the same multi-part geometry (`level_0=0`), which had three parts (`level_1=0,1,2`). Figure 4.28 demonstrates the effect of `.explode` in converting a layer with multi-part geometries into a layer with single-part geometries.

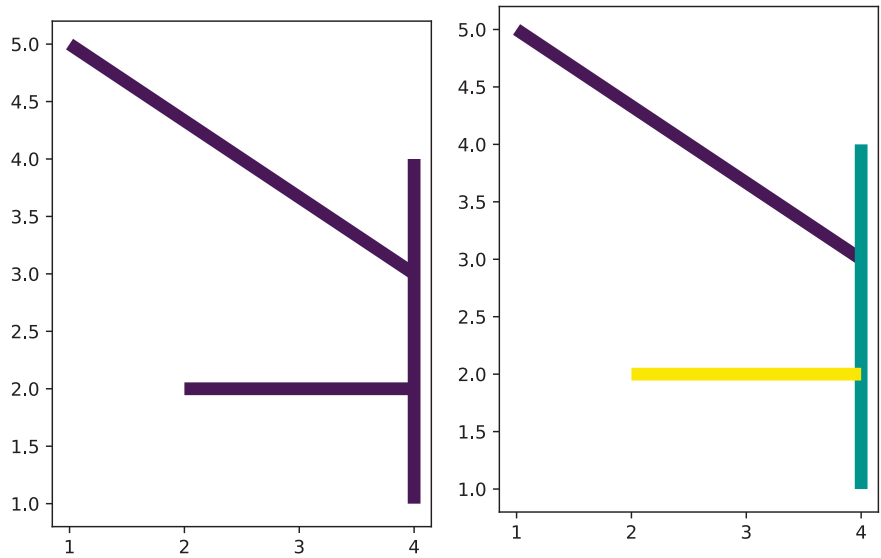
```
dat.plot(column='id', linewidth=7);
dat1.plot(column='level_1', linewidth=7);
```

As a side-note, let's demonstrate how the above **shapely** casting methods can be translated to **geopandas**. Suppose that we want to transform `dat1`, which is a layer of type `'LineString'` with three features, to a layer of type `'MultiPoint'` (also with three features). Recall that for a single geometry, we use the expression `shapely.MultiPoint(x.coords)`, where `x` is a `'LineString'` (Figure 4.22). When dealing with a `GeoDataFrame`, we wrap the conversion into `.apply`, to apply it to all geometries:

```
dat2 = dat1.copy()
dat2.geometry = dat2.geometry.apply(lambda x: shapely.MultiPoint(x.coords))
dat2
```

	level_0	level_1	id	geometry
0	0	0	1	MULTIPOINT (1 5, 4 3)
1	0	1	1	MULTIPOINT (4 4, 4 1)
2	0	2	1	MULTIPOINT (2 2, 4 2)

The result is illustrated in Figure 4.29.



(a) `'MultiLineString'` layer

(b) `'LineString'` layer, after applying `.explode`

Figure 4.28: Transformation of a `'MultiLineString'` layer with one feature, into a `'LineString'` layer with three features, using `.explode`

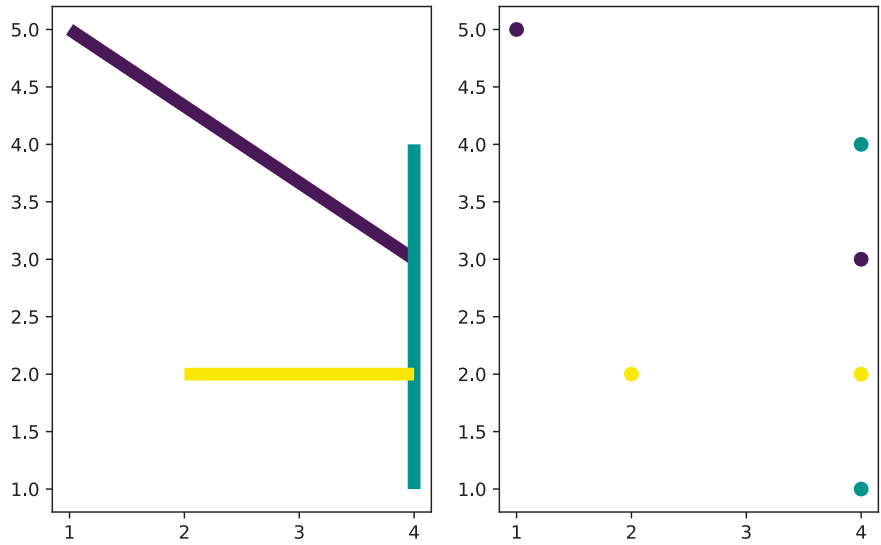
```
dat1.plot(column='level_1', linewidth=7);
dat2.plot(column='level_1', markersize=50);
```

The opposite transformation, i.e., 'single-part to multi-part', is achieved using the `.dissolve` method (which we are already familiar with, see Section 4.2.7). For example, here is how we can get from the `'LineString'` layer with three features back to the `'MultiLineString'` layer with one feature (since, in this case, there is just one group):

```
dat1.dissolve(by='id').reset_index()
```

	id	geometry	level_0	level_1
0	1	MULTILINESTRING ((1 5, 4 3), (4...	0	0

The next code chunk is another example, dissolving the 16 polygons in `nz` into two geometries of the north and south parts (i.e., the two `'Island'` groups).



(a) `'LineString'` layer (b) `'MultiPoint'` layer

Figure 4.29: Transformation of a `'LineString'` layer with three features, into a `'MultiPoint'` layer (also with three features), using `.apply` and **shapely** methods

```
nz_dis1 = nz[['Island', 'Population', 'geometry']] \
    .dissolve(by='Island', aggfunc='sum') \
    .reset_index()
nz_dis1
```

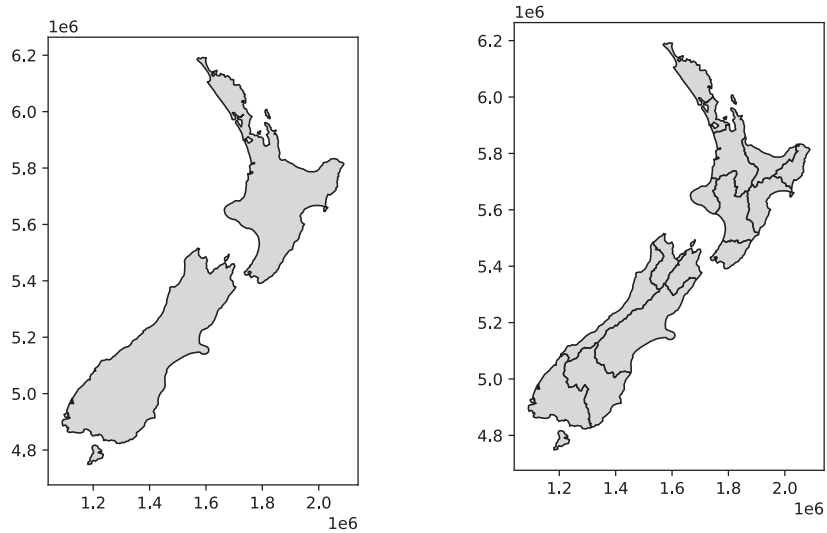
	Island	geometry	Population
0	North	MULTIPOLYGON (((1865558.829 546...	3671600.0
1	South	MULTIPOLYGON (((1229729.735 479...	1115600.0

Note that `.dissolve` not only combines single-part into multi-part geometries, but also dissolves any internal borders. So, in fact, the resulting geometries may be single-part (in case when all parts touch each other, unlike in `nz`). If, for some reason, we want to combine geometries into multi-part *without* dissolving, we can fall back to the **pandas** `.agg` method (custom table aggregation), supplemented with a **shapely** function specifying how exactly we want to transform each group of geometries into a new single geometry. In the following example, for instance, we collect all `'Polygon'` and `'MultiPolygon'` parts of `nz` into a single `'MultiPolygon'` geometry with many separate parts (i.e., without dissolving), per group.

```
nz_dis2 = nz \
    .groupby('Island') \
    .agg({
        'Population': 'sum',
        'geometry': lambda x: shapely.MultiPolygon(x.explode().to_list())
    }) \
    .reset_index()
nz_dis2 = gpd.GeoDataFrame(nz_dis2).set_geometry('geometry').set_crs(nz.crs)
nz_dis2
```

	Island	Population	geometry
0	North	3671600.0	MULTIPOLYGON (((1745493.196 600...
1	South	1115600.0	MULTIPOLYGON (((1557042.169 531...

The difference between the last two results `nz_dis1` and `nz_dis2` (with and without dissolving, respectively) is not evident in the printout: in both cases we got a layer with two features of type `'MultiPolygon'`. However, in the first case internal borders were dissolved, while in the second case they were not. This is illustrated in Figure 4.30:



(a) Dissolving (using the **geopandas** `.dissolve` method)

(b) Combining into multi-part without dissolving (using `.agg` and a custom **shapely**-based function)

Figure 4.30: Combining New Zealand geometries into one, for each island, with and without dissolving

```
nz_dis1.plot(color='lightgrey', edgecolor='black');
nz_dis2.plot(color='lightgrey', edgecolor='black');
```

It is also worthwhile to note the `.boundary` and `.exterior` properties of `GeoSeries`, which are used to cast polygons to lines, with or without interior rings, respectively (see Section 5.4.2).

4.3 Geometric operations on raster data

Geometric raster operations include the shift, flipping, mirroring, scaling, rotation, or warping of images. These operations are necessary for a variety of applications including georeferencing, used to allow images to be overlaid on an accurate map with a known CRS (Liu and Mason 2009). A variety of georeferencing techniques exist, including:

- Georectification based on known ground control points
- Orthorectification, which also accounts for local topography
- Image registration is used to combine images of the same thing but shot from different sensors, by aligning one image with another (in terms of coordinate system and resolution)

Python is rather unsuitable for the first two points since these often require manual intervention which is why they are usually done with the help of dedicated GIS software. On the other hand, aligning several images is possible in Python and this section shows among others how to do so. This often includes changing the extent, the resolution, and the origin of an image. A matching projection is of course also required but is already covered in Section 6.8.

In any case, there are other reasons to perform a geometric operation on a single raster image. For instance, a common reason for aggregating a raster is to decrease run-time or save disk space. Of course, this approach is only recommended if the task at hand allows a coarser resolution of raster data.

4.3.1 Extent and origin

When merging or performing map algebra on rasters, their resolution, projection, origin, and/or extent have to match. Otherwise, how should we add the values of one raster with a resolution of `0.2` decimal degree to a second raster with a resolution of `1` decimal degree? The same problem arises when we would like to merge satellite imagery from different sensors with different projections and resolutions. We can deal with such mismatches by aligning the rasters. Typically, raster alignment is done through resampling—that way, it is guaranteed that the rasters match exactly (Section 4.3.3). However, sometimes

it can be useful to modify raster placement and extent manually, by adding or removing rows and columns, or by modifying the origin, that is, slightly shifting the raster. Sometimes, there are reasons other than alignment with a second raster for manually modifying raster extent and placement. For example, it may be useful to add extra rows and columns to a raster prior to focal operations, so that it is easier to operate on the edges.

Let's demonstrate the first operation, raster padding. First, we will read the array with the `elev.tif` values:

```
r = src_elev.read(1)
r
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35, 36]], dtype=uint8)
```

To pad an `ndarray`, we can use the `np.pad` function. The function accepts an array, and a tuple of the form `((rows_top,rows_bottom),(columns_left, columns_right))`. Also, we can specify the value that's being used for padding with `constant_values` (e.g., `18`). For example, here we pad `r` with one extra row and two extra columns, on both sides, resulting in the array `r_pad`:

```
rows = 1
cols = 2
r_pad = np.pad(r, ((rows,rows),(cols,cols)), constant_values=18)
r_pad
```

```
array([[18, 18, 18, 18, 18, 18, 18, 18, 18, 18],
       [18, 18,  1,  2,  3,  4,  5,  6, 18, 18],
       [18, 18,  7,  8,  9, 10, 11, 12, 18, 18],
       [18, 18, 13, 14, 15, 16, 17, 18, 18, 18],
       [18, 18, 19, 20, 21, 22, 23, 24, 18, 18],
       [18, 18, 25, 26, 27, 28, 29, 30, 18, 18],
       [18, 18, 31, 32, 33, 34, 35, 36, 18, 18],
       [18, 18, 18, 18, 18, 18, 18, 18, 18, 18]], dtype=uint8)
```

However, for `r_pad` to be used in any spatial operation, we also have to update its transformation matrix. Whenever we add extra columns on the left, or extra rows on top, the raster *origin* changes. To reflect this fact, we have to take to 'original' origin and add the required multiple of pixel widths or heights (i.e., raster resolution steps). The transformation matrix of a raster is accessible from the raster file metadata (Section 1.3.2) or, as a shortcut, through the `.transform` property of the raster file connection. For example, the next code chunk shows the transformation matrix of `elev.tif`.

```
src_elev.transform
```

```
Affine(0.5, 0.0, -1.5,
       0.0, -0.5, 1.5)
```

From the transformation matrix, we are able to extract the origin.

```
xmin, ymax = src_elev.transform[2], src_elev.transform[5]
xmin, ymax
```

```
(-1.5, 1.5)
```

We can also get the resolution of the data, which is the distance between two adjacent pixels.

```
dx, dy = src_elev.transform[0], src_elev.transform[4]
dx, dy
```

```
(0.5, -0.5)
```

These two parts of information are enough to calculate the new origin (`xmin_new`,`ymax_new`) of the padded raster.

```
xmin_new = xmin - dx * cols
ymax_new = ymax - dy * rows
xmin_new, ymax_new
```

```
(-2.5, 2.0)
```

Using the updated origin, we can update the transformation matrix (Section 1.3.2). Keep in mind that the meaning of the last two arguments is `xsize`, `ysize`, so we need to pass the absolute value of `dy` (since it is negative).

```
new_transform = rasterio.transform.from_origin(
    west=xmin_new,
    north=ymax_new,
    xsize=dx,
    ysize=abs(dy)
)
new_transform
```

```
Affine(0.5, 0.0, -2.5,
       0.0, -0.5, 2.0)
```

Figure 4.31 shows the padded raster, with the outline of the original `elev.tif` (in red), demonstrating that the origin was shifted correctly and the `new_transform` works fine.

```
fig, ax = plt.subplots()
rasterio.plot.show(r_pad, transform=new_transform, cmap='Greys', ax=ax)
elev_bbox = gpd.GeoSeries(shapely.box(*src_elev.bounds))
elev_bbox.plot(color='none', edgecolor='red', ax=ax);
```

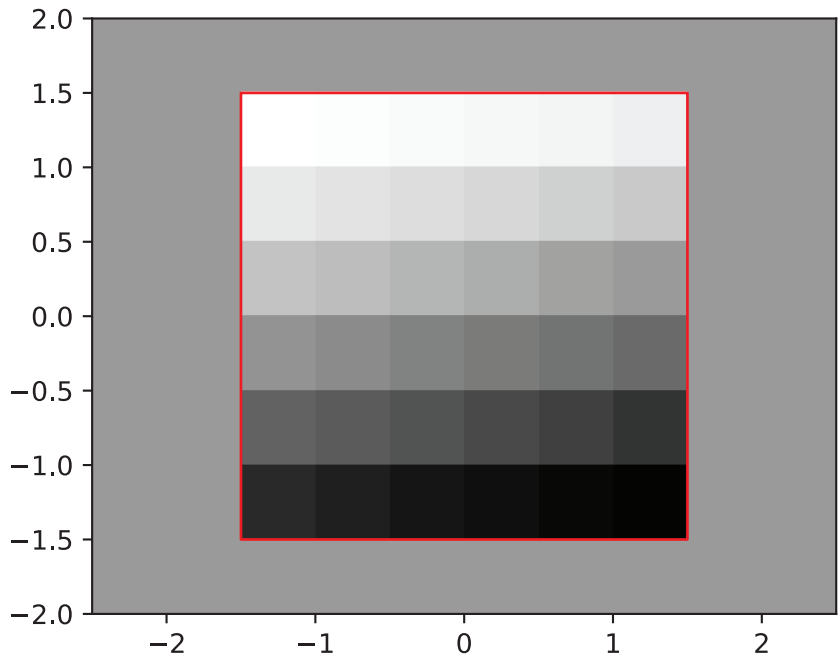


Figure 4.31: The padded `elev.tif` raster, and the extent of the original `elev.tif` raster (in red)

We can shift a raster origin not just when padding, but in any other use case, just by changing its transformation matrix. The effect is that the raster is going to be shifted (which is analogous to `.translate` for shifting a vector layer, see Section 4.2.4). Manually shifting a raster to arbitrary distance is rarely needed in real-life scenarios, but it is useful to know how to do it at least for a better understanding of the concept of *raster origin*. As an example, let's shift the origin of `elev.tif` by `(-0.25,0.25)`. First, we need to calculate the new origin.

```
xmin_new = xmin - 0.25  # shift xmin to the left
ymax_new = ymax + 0.25  # shift ymax upwards
xmin_new, ymax_new
```

```
(-1.75, 1.75)
```

To shift the origin in other directions we should change the two operators (`-`, `+`) accordingly.

Then, same as when padding (see above), we create an updated transformation matrix.

```
new_transform = rasterio.transform.from_origin(
    west=xmin_new,
    north=ymax_new,
    xsize=dx,
    ysize=abs(dy)
)
new_transform
```

```
Affine(0.5, 0.0, -1.75,
       0.0, -0.5, 1.75)
```

Figure 4.32 shows the shifted raster and the outline of the original `elev.tif` raster (in red).

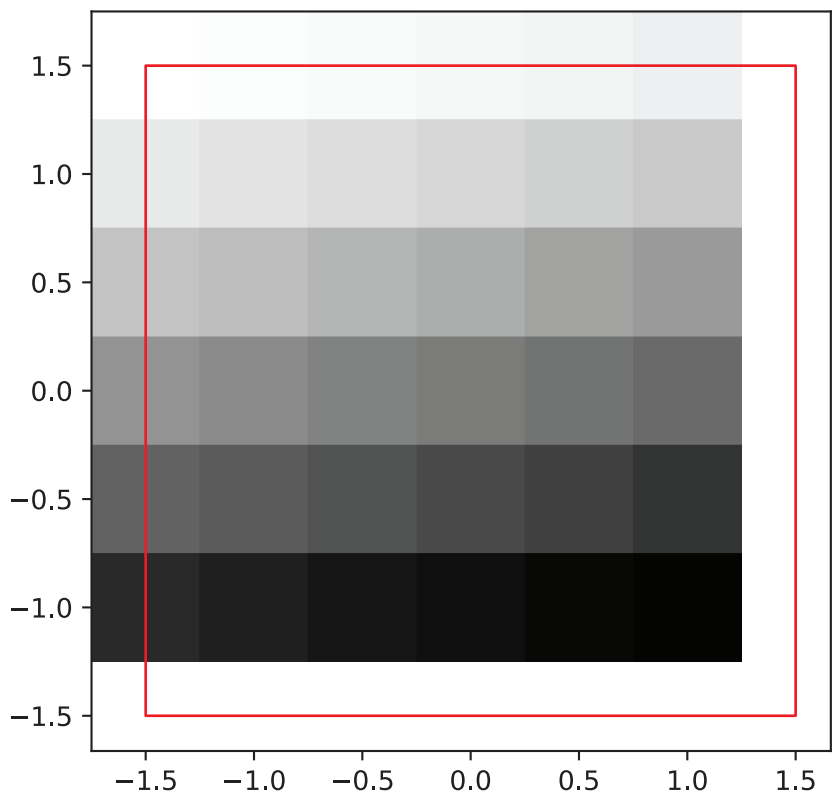


Figure 4.32: The `elev.tif` raster shifted by `(0.25,0.25)`, and its original extent (in red)

```
fig, ax = plt.subplots()
rasterio.plot.show(r, transform=new_transform, cmap='Greys', ax=ax)
elev_bbox.plot(color='none', edgecolor='red', ax=ax);
```

4.3.2 Aggregation and disaggregation

Raster datasets vary based on their resolution, from high-resolution datasets that enable individual trees to be seen, to low-resolution datasets covering large swaths of the Earth. Raster datasets can be transformed to either decrease (aggregate) or increase (disaggregate) their resolution, for a number of reasons. For example, aggregation can be used to reduce computational resource requirements of raster storage and subsequent steps, while disaggregation can be used to match other datasets, or to add detail.

> **Note**
>
> Raster aggregation is, in fact, a special case of raster resampling (see Section 4.3.3), where the target raster grid is aligned with the original raster, only with coarser pixels. Conversely, raster resampling is the general case where the new grid is not necessarily an aggregation of the original one, but any other type of grid (i.e., shifted and or having increased/reduced resolution, by any factor).

As an example, we here change the spatial resolution of `dem.tif` by a factor of `5` (Figure 4.33). To aggregate a raster using **rasterio**, we go through two steps:

- Reading the raster values (using `.read`) into an `out_shape` that is different from the original `.shape`
- Updating the `transform` according to `out_shape`

Let's demonstrate it, using the `dem.tif` file. Note the original shape of the raster; it has `117` rows and `117` columns.

```
src.read(1).shape
```

```
(117, 117)
```

Also note the transform, which tells us that the raster resolution is about 30.85 m.

```
src.transform
```

```
Affine(30.849999999999604, 0.0, 794599.1076146346,
       0.0, -30.84999999999363, 8935384.324602526)
```

To aggregate, instead of reading the raster values the usual way, as in `src.read(1)`, we can specify `out_shape` to read the values into a different shape. Here, we calculate a new shape which is downscaled by a factor of `5`, i.e., the number of rows and columns is multiplied by `0.2`. We must truncate any partial rows and columns, e.g., using `int`. Each new pixel is now obtained, or resampled, from $\sim 5 \times 5 = \sim 25$ 'old' raster values. It is crucial to choose an appropriate *resampling method* through the `resampling` parameter. Here we use `rasterio.enums.Resampling.average`, i.e., the new 'large' pixel value is the average of all coinciding small pixels, which makes sense for our elevation data in `dem.tif`. See Section 4.3.3 for a list of other available methods.

```
factor = 0.2
r = src.read(1,
    out_shape=(
        int(src.height * factor),
        int(src.width * factor)
        ),
    resampling=rasterio.enums.Resampling.average
)
```

As expected, the resulting array `r` has ~5 times smaller dimensions, as shown below.

```
r.shape
```

```
(23, 23)
```

What's left to be done is the second step, to update the transform, taking into account the change in raster shape. This can be done as follows, using `.transform.scale`.

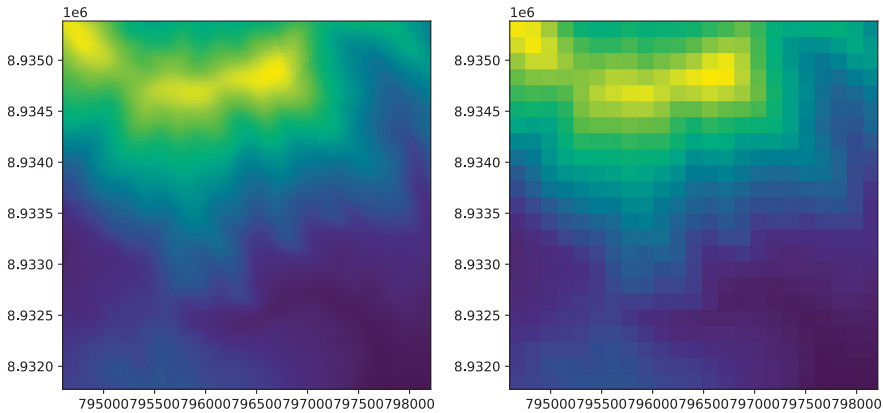
```
new_transform = src.transform * src.transform.scale(
    (src.width / r.shape[1]),
    (src.height / r.shape[0])
)
new_transform
```

```
Affine(156.93260869565017, 0.0, 794599.1076146346,
       0.0, -156.9326086956198, 8935384.324602526)
```

Figure 4.33 shows the original raster and the aggregated one.

```
rasterio.plot.show(src);
rasterio.plot.show(r, transform=new_transform);
```

This is a good opportunity to demonstrate exporting a raster with modified dimensions and transformation matrix. We can update the raster metadata required for writing with the `update` method.



(a) Original (b) Aggregated (using average resampling)

Figure 4.33: Aggregating a raster by a factor of 5, using average resampling

```
dst_kwargs = src.meta.copy()
dst_kwargs.update({
    'transform': new_transform,
    'width': r.shape[1],
    'height': r.shape[0],
})
dst_kwargs
```

```
{'driver': 'GTiff',
 'dtype': 'float32',
 'nodata': nan,
 'width': 23,
 'height': 23,
 'count': 1,
 'crs': CRS.from_epsg(32717),
 'transform': Affine(156.93260869565017, 0.0, 794599.1076146346,
        0.0, -156.9326086956198, 8935384.324602526)}
```

Then we can create a new file (`dem_agg5.tif`) in writing mode, and write the values from the aggregated array `r` into the 1st band of the file (see Section 7.6.2 for a detailed explanation of writing raster files with **rasterio**).

```
dst = rasterio.open('output/dem_agg5.tif', 'w', **dst_kwargs)
dst.write(r, 1)
dst.close()
```

Note

The `**` syntax in Python is known as variable-length '*keyword* arguments'. It is used to pass a dictionary of numerous `parameter:argument` pairs to named arguments of a function. In `rasterio.open` writing mode, the 'keyword arguments' syntax often comes in handy, because, instead of specifying each and every property of a new file, we pass a (modified) `.meta` dictionary based on another, template, raster.
Technically, keep in mind that the expression:

```
rasterio.open('out.tif', 'w', **dst_kwargs)
```

where `dst_kwargs` is a `dict` of the following form (typically coming from a template raster, possibly with few updated properties using `.update`, see above):

```
{'driver': 'GTiff',
 'dtype': 'float32',
 'nodata': nan,
 ...
}
```

is a shortcut of:

```
rasterio.open(
    'out.tif', 'w',
    driver=dst_kwargs['driver'],
    dtype=dst_kwargs['dtype'],
    nodata=dst_kwargs['nodata'],
    ...
)
```

Positional arguments is a related technique; see note in Section 6.8.

The opposite operation, namely disaggregation, is when we increase the resolution of raster objects. Either of the supported resampling methods (see Section 4.3.3) can be used. However, since we are not actually summarizing information but transferring the value of a large pixel into multiple small pixels, it makes sense to use either:

- Nearest neighbor resampling (`rasterio.enums.Resampling.nearest`), when we want to keep the original values as is, since modifying them would be incorrect (such as in categorical rasters)
- Smoothing techniques, such as bilinear resampling (`rasterio.enums.Resampling.bilinear`), when we would like the smaller pixels to reflect gradual change between the original values, e.g., when the disaggregated raster is used for visualization purposes

To disaggregate a raster, we go through exactly the same workflow as for aggregation, only using a different scaling factor, such as `factor=5` instead of `factor=0.2`, i.e., *increasing* the number of raster pixels instead of decreasing.

In the example below, we disaggregate using bilinear interpolation, to get a smoothed high-resolution raster.

```
factor = 5
r2 = src.read(1,
    out_shape=(
        int(src.height * factor),
        int(src.width * factor)
        ),
    resampling=rasterio.enums.Resampling.bilinear
)
```

As expected, the dimensions of the disaggregated raster are this time ~5 times *bigger* than the original ones.

```
r2.shape
```

```
(585, 585)
```

To calculate the new transform, we use the same expression as for aggregation, only with the new `r2` shape.

```
new_transform2 = src.transform * src.transform.scale(
    (src.width / r2.shape[1]),
    (src.height / r2.shape[0])
)
new_transform2
```

```
Affine(6.169999999999921, 0.0, 794599.1076146346,
       0.0, -6.169999999998726, 8935384.324602526)
```

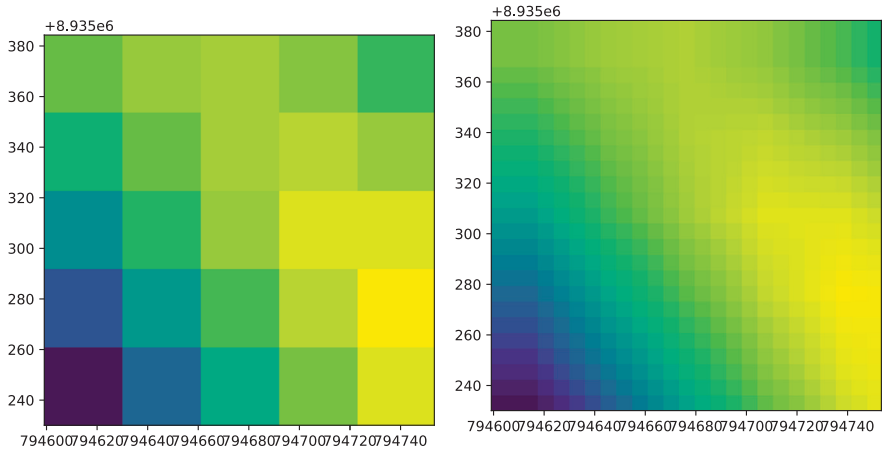
The original raster `dem.tif` was already quite detailed, so it would be difficult to see any difference when plotting it along with the disaggregation result. A zoom-in of a small section of the rasters works better. Figure 4.34 shows the top-left corners of the original raster and the disaggregated one, demonstrating the increase in the number of pixels through disaggregation.

```
rasterio.plot.show(src.read(1)[:5, :5], transform=src.transform);
rasterio.plot.show(r2[:25, :25], transform=new_transform2);
```

Code to export the disaggregated raster would be identical to the one used above for the aggregated raster.

4.3.3 Resampling

Raster aggregation and disaggregation (Section 4.3.2) are only suitable when we want to change just the resolution of our raster by a fixed factor. However, what to do when we have two or more rasters with different resolutions and origins? This is the role of resampling—a process of computing values for new



(a) Original

(b) Disaggregated (using bilinear resampling)

Figure 4.34: Disaggregating a raster by a factor of 5, using bilinear resampling. Only a small portion (top-left corner) of the rasters is shown, to zoom-in and demonstrate the effect of disaggregation.

pixel locations. In short, this process takes the values of our original raster and recalculates new values for a target raster with custom resolution and origin (Figure 4.35).

There are several methods for estimating values for a raster with different resolutions/origins (Figure 4.35). The main resampling methods include:

- Nearest neighbor—assigns the value of the nearest cell of the original raster to the cell of the target one. This is a fast simple technique that is usually suitable for resampling categorical rasters
- Bilinear interpolation—assigns a weighted average of the four nearest cells from the original raster to the cell of the target one. This is the fastest method that is appropriate for continuous rasters
- Cubic interpolation—uses values of the 16 nearest cells of the original raster to determine the output cell value, applying third-order polynomial functions. Used for continuous rasters and results in a smoother surface compared to bilinear interpolation, but is computationally more demanding
- Cubic spline interpolation—also uses values of the 16 nearest cells of the original raster to determine the output cell value, but applies cubic splines (piecewise third-order polynomial functions). Used for continuous rasters
- Lanczos windowed sinc resampling—uses values of the 36 nearest cells of the original raster to determine the output cell value. Used for continuous rasters

- Additionally, we can use straightforward summary methods, taking into account all pixels that coincide with the target pixel, such as average (Figure 4.33), minimum, maximum (Figure 4.35), median, mode, and sum

The above explanation highlights that only nearest neighbor resampling is suitable for categorical rasters, while all remaining methods can be used (with different outcomes) for continuous rasters.

With **rasterio**, resampling can be done using the `rasterio.warp.reproject` function. To clarify this naming convention, note that raster *reprojection* is not fundamentally different from *resampling*—the difference is just whether the target grid is in the same CRS as the origin (resampling) or in a different CRS (reprojection). In other words, reprojection is *resampling* into a grid that is in a different CRS. Accordingly, both resampling and reprojection are done using the same function `rasterio.warp.reproject`. We will demonstrate *reprojection* using `rasterio.warp.reproject` later in Section 6.8.

The information required for `rasterio.warp.reproject`, whether we are resampling or reprojecting, is:

- The source and target *CRS*. These may be identical, when resampling, or different, when reprojecting
- The source and target *transform*

Importantly, `rasterio.warp.reproject` can work with file connections, such as a connection to an output file in write (`'w'`) mode. This makes the function efficient for large rasters.

The target and destination CRS are straightforward to specify, depending on our choice. The source transform is also readily available, through the `.transform` property of the source file connection. The only complicated part is to figure out the *destination transform*. When resampling, the transform is typically derived either from a *template* raster, such as an existing raster file that we would like our origin raster to match, or from a numeric specification of our target grid (see below). Otherwise, when the exact grid is not of importance, we can simply aggregate or disaggregate our raster as shown above (Section 4.3.2). (Note that when *reprojecting*, the target transform is more difficult to figure out, therefore we further use the `rasterio.warp.calculate_default_transform` function to compute it, as will be shown in Section 6.8.)

Finally, the resampling method is specified through the `resampling` parameter of `rasterio.warp.reproject`. The default is nearest neighbor resampling. However, as mentioned above, you should be aware of the distinction between resampling methods, and choose the appropriate one according to the data type (continuous/categorical), the input and output resolution, and resampling purposes. Possible arguments for `resampling` include:

- `rasterio.enums.Resampling.nearest`—Nearest neighbor
- `rasterio.enums.Resampling.bilinear`—Bilinear

- `rasterio.enums.Resampling.cubic`—Cubic
- `rasterio.enums.Resampling.lanczos`—Lanczos windowed
- `rasterio.enums.Resampling.average`—Average
- `rasterio.enums.Resampling.mode`—Mode. i.e., most common value
- `rasterio.enums.Resampling.min`—Minimum
- `rasterio.enums.Resampling.max`—Maximum
- `rasterio.enums.Resampling.med`—Median
- `rasterio.enums.Resampling.sum`—Sum

Let's demonstrate resampling into a destination grid which is specified through numeric constraints, such as the extent and resolution. Again, these could have been specified manually (such as here), or obtained from a template raster metadata that we would like to match. Note that the resolution of the destination grid is ~10 times more coarse (300 m) than the original resolution of `dem.tif` (~30 m) (Figure 4.35).

```
xmin = 794650
xmax = 798250
ymin = 8931750
ymax = 8935350
res = 300
```

The corresponding transform based on these constraints can be created using the `rasterio.transform.from_origin` function, as follows:

```
dst_transform = rasterio.transform.from_origin(
    west=xmin,
    north=ymax,
    xsize=res,
    ysize=res
)
dst_transform
```

```
Affine(300.0, 0.0, 794650.0,
       0.0, -300.0, 8935350.0)
```

In case we needed to resample into a grid specified by an existing template raster, we could have skipped this step and simply read the transform from the template file, as in `rasterio.open('template.tif').transform`.

We can move on to creating the destination file connection. For that, we also have to know the raster dimensions, which can be derived from the extent and the resolution.

```
width = int((xmax - xmin) / res)
height = int((ymax - ymin) / res)
width, height
```

```
(12, 12)
```

Now we can create the destination file connection. We are using the same metadata as the source file, except for the dimensions and the transform, which are going to be different and reflect the resampling process.

```
dst_kwargs = src.meta.copy()
dst_kwargs.update({
    'transform': dst_transform,
    'width': width,
    'height': height
})
dst = rasterio.open('output/dem_resample_nearest.tif', 'w', **dst_kwargs)
```

Finally, we reproject using function `rasterio.warp.reproject`. Note that the source and destination are specified using `rasterio.band` applied on both file connections, reflecting the fact that we operate on a specific layer of the rasters. The resampling method being used here is nearest neighbor resampling (`rasterio.enums.Resampling.nearest`).

```
rasterio.warp.reproject(
    source=rasterio.band(src, 1),
    destination=rasterio.band(dst, 1),
    src_transform=src.transform,
    src_crs=src.crs,
    dst_transform=dst_transform,
    dst_crs=src.crs,
    resampling=rasterio.enums.Resampling.nearest
)
```

```
(Band(ds=<open DatasetWriter name='output/dem_resample_nearest.tif' mode='w'>,
      bidx=1, dtype='float32', shape=(12, 12)),
 Affine(300.0, 0.0, 794650.0,
        0.0, -300.0, 8935350.0))
```

In the end, we close the file connection, thus finalizing the new file `output/dem_resample_nearest.tif` with the resampling result (Figure 4.35).

```
dst.close()
```

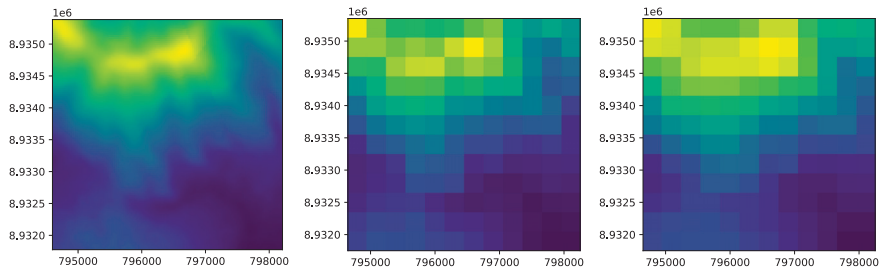
Here is another code section just to demonstrate a different resampling method, the maximum resampling, i.e., every new pixel gets the maximum value of all the original pixels it coincides with (Figure 4.35). Note that all arguments in the `rasterio.warp.reproject` function call are identical to the previous example, except for the `resampling` method.

```
dst = rasterio.open('output/dem_resample_maximum.tif', 'w', **dst_kwargs)
rasterio.warp.reproject(
    source=rasterio.band(src, 1),
    destination=rasterio.band(dst, 1),
    src_transform=src.transform,
    src_crs=src.crs,
    dst_transform=dst_transform,
    dst_crs=src.crs,
```

```
    resampling=rasterio.enums.Resampling.max
)
dst.close()
```

The original raster `dem.tif`, and the two resampling results `dem_resample_nearest.tif` and `dem_resample_maximum.tif`, are shown in Figure 4.35.

```
# Input
fig, ax = plt.subplots(figsize=(4,4))
rasterio.plot.show(src, ax=ax);
# Nearest neighbor
fig, ax = plt.subplots(figsize=(4,4))
rasterio.plot.show(rasterio.open('output/dem_resample_nearest.tif'), ax=ax);
# Maximum
fig, ax = plt.subplots(figsize=(4,4))
rasterio.plot.show(rasterio.open('output/dem_resample_maximum.tif'), ax=ax);
```



(a) Input (b) Nearest neighbor (c) Maximum

Figure 4.35: The original raster `dem.tif` and two different resampling method results

5

Raster-vector interactions

Prerequisites

This chapter requires importing the following packages:

```
import os
import math
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import shapely
import geopandas as gpd
import rasterio
import rasterio.plot
import rasterio.mask
import rasterio.features
import rasterstats
```

It also relies on the following data files:

```
src_srtm = rasterio.open('data/srtm.tif')
src_nlcd = rasterio.open('data/nlcd.tif')
src_grain = rasterio.open('output/grain.tif')
src_elev = rasterio.open('output/elev.tif')
src_dem = rasterio.open('data/dem.tif')
zion = gpd.read_file('data/zion.gpkg')
zion_points = gpd.read_file('data/zion_points.gpkg')
cycle_hire_osm = gpd.read_file('data/cycle_hire_osm.gpkg')
us_states = gpd.read_file('data/us_states.gpkg')
nz = gpd.read_file('data/nz.gpkg')
src_nz_elev = rasterio.open('data/nz_elev.tif')
```

DOI: 10.1201/9781003379911-5

5.1 Introduction

This chapter focuses on interactions between raster and vector geographic data models, both introduced in Chapter 1. It includes three main techniques:

- Raster cropping and masking using vector objects (Section 5.2)
- Extracting raster values using different types of vector data (Section 5.3)
- Raster-vector conversion (Section 5.4 and Section 5.5)

These concepts are demonstrated using data from previous chapters, to understand their potential real-world applications.

5.2 Raster masking and cropping

Many geographic data projects involve integrating data from many different sources, such as remote sensing images (rasters) and administrative boundaries (vectors). Often the extent of input raster datasets is larger than the area of interest. In this case, raster *masking*, *cropping*, or both, are useful for unifying the spatial extent of input data (Figure 5.2 (b) and (c), and the following two examples, illustrate the difference between masking and cropping). Both operations reduce object memory use and associated computational resources for subsequent analysis steps, and may be a necessary preprocessing step before creating attractive maps involving raster data.

We will use two layers to illustrate raster cropping:

- The `srtm.tif` raster representing elevation, in meters above sea level, in south-western Utah: a **rasterio** file connection named `src_srtm` (see Figure 5.2 (a))
- The `zion.gpkg` vector layer representing the Zion National Park boundaries (a `GeoDataFrame` named `zion`)

Both target and cropping objects must have the same projection. Since it is easier and more precise to reproject vector layers, compared to rasters, we use the following expression to reproject (Section 6.7) the vector layer `zion` into the CRS of the raster `src_srtm`.

```
zion = zion.to_crs(src_srtm.crs)
```

To mask the image, i.e., convert all pixels which do not intersect with the `zion` polygon to 'No Data', we use the `rasterio.mask.mask` function.

```
out_image_mask, out_transform_mask = rasterio.mask.mask(
    src_srtm,
    zion.geometry,
    crop=False,
    nodata=9999
)
```

Note that we need to choose and specify a 'No Data' value, within the valid range according to the data type. Since `srtm.tif` is of type `uint16` (how can we check?), we choose `9999` (a positive integer that is guaranteed not to occur in the raster). Also note that **rasterio** does not directly support **geopandas** data structures, so we need to pass a 'collection' of **shapely** geometries: a `GeoSeries` (see above) or a `list` of **shapely** geometries (see next example) both work. The output consists of two objects. The first one is the `out_image` array with the masked values.

```
out_image_mask
```

```
array([[[9999, 9999, 9999, ..., 9999, 9999, 9999],
        [9999, 9999, 9999, ..., 9999, 9999, 9999],
        [9999, 9999, 9999, ..., 9999, 9999, 9999],
        ...,
        [9999, 9999, 9999, ..., 9999, 9999, 9999],
        [9999, 9999, 9999, ..., 9999, 9999, 9999],
        [9999, 9999, 9999, ..., 9999, 9999, 9999]]], dtype=uint16)
```

The second one is a new transformation matrix `out_transform`.

```
out_transform_mask
```

```
Affine(0.0008333333332777796, 0.0, -113.23958321278403,
       0.0, -0.0008333333332777843, 37.512916763165805)
```

Note that masking (without cropping!) does not modify the raster extent. Therefore, the new transform is identical to the original (`src_srtm.transform`).

Unfortunately, the `out_image` and `out_transform` objects do not contain any information indicating that `9999` represents 'No Data'. To associate the information with the raster, we must write it to file along with the corresponding metadata. For example, to write the masked raster to file, we first need to modify the 'No Data' setting in the metadata.

```
dst_kwargs = src_srtm.meta
dst_kwargs.update(nodata=9999)
dst_kwargs
```

```
{'driver': 'GTiff',
 'dtype': 'uint16',
 'nodata': 9999,
 'width': 465,
 'height': 457,
 'count': 1,
 'crs': CRS.from_epsg(4326),
 'transform': Affine(0.0008333333332777796, 0.0, -113.23958321278403,
        0.0, -0.0008333333332777843, 37.512916763165805)}
```

Then we can write the masked raster to file with the updated metadata object.

```
new_dataset = rasterio.open('output/srtm_masked.tif', 'w', **dst_kwargs)
new_dataset.write(out_image_mask)
new_dataset.close()
```

Now we can re-import the raster and check that the 'No Data' value is correctly set.

```
src_srtm_mask = rasterio.open('output/srtm_masked.tif')
```

The `.meta` property contains the `nodata` entry. Now, any relevant operation (such as plotting, see Figure 5.2 (b)) will take 'No Data' into account.

```
src_srtm_mask.meta
```

```
{'driver': 'GTiff',
 'dtype': 'uint16',
 'nodata': 9999.0,
 'width': 465,
 'height': 457,
 'count': 1,
 'crs': CRS.from_epsg(4326),
 'transform': Affine(0.0008333333332777796, 0.0, -113.23958321278403,
        0.0, -0.0008333333332777843, 37.512916763165805)}
```

The related operation, cropping, reduces the raster extent to the extent of the vector layer:

- To crop *and* mask, we can use `rasterio.mask.mask`, same as above for masking, while setting `crop=True` (Figure 5.2 (d))
- To just crop, *without* masking, we can derive the bounding box polygon of the vector layer, and then crop using that polygon, also combined with `crop=True` (Figure 5.2 (c))

For the example of cropping only, the extent polygon of `zion` can be obtained as a `shapely` geometry object using `.union_all().envelope` (Figure 5.1).

```
bb = zion.union_all().envelope
bb
```

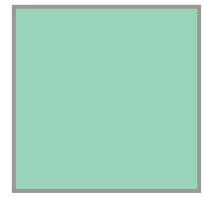


Figure 5.1: Bounding box `'Polygon'` geometry of the `zion` layer

The extent can now be used for masking. Here, we are also using the `all_touched=True` option, so that pixels which are partially overlapping with the extent are also included in the output.

```
out_image_crop, out_transform_crop = rasterio.mask.mask(
    src_srtm,
    [bb],
    crop=True,
    all_touched=True,
    nodata=9999
)
```

In the case of cropping, there is no particular reason to write the result to file for easier plotting, such as in the other two examples, since there are no 'No Data' values (Figure 5.2 (c)).

> **Note**
>
> As mentioned above, **rasterio** functions typically accept vector geometries in the form of `lists` of `shapely` objects. `GeoSeries` are conceptually very similar, and also accepted. However, even an individual geometry has to be in a `list`, which is why we pass `[bb]`, and not `bb`, in the above `rasterio.mask.mask` function call (the latter would raise an error).

Finally, the third example is where we perform both crop and mask operations, using `rasterio.mask.mask` with `crop=True` passing `zion.geometry`.

```
out_image_mask_crop, out_transform_mask_crop = rasterio.mask.mask(
    src_srtm,
    zion.geometry,
    crop=True,
    nodata=9999
)
```

When writing the result to a file, it is here crucial to update the transform and dimensions, since they were modified as a result of cropping. Also note that `out_image_mask_crop` is a three-dimensional array (even though it has one band in this case), so the number of rows and columns are in `.shape[1]` and `.shape[2]` (rather than `.shape[0]` and `.shape[1]`), respectively.

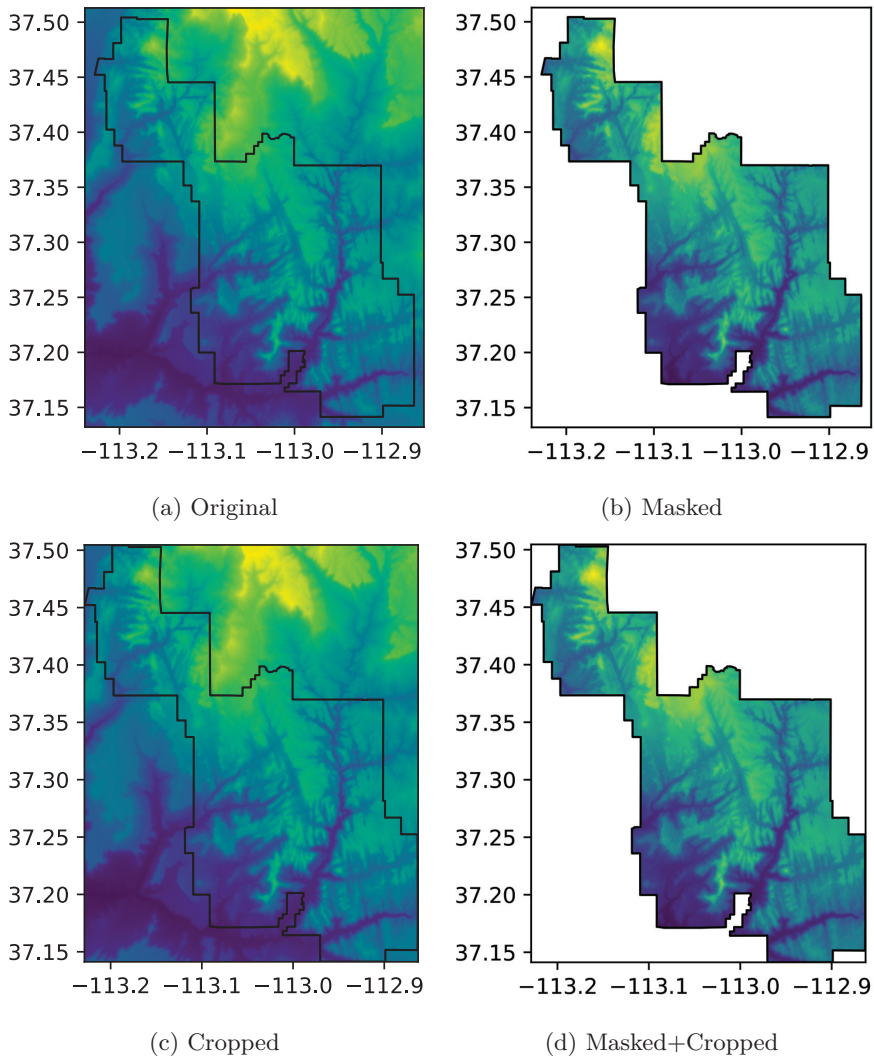
```
dst_kwargs = src_srtm.meta
dst_kwargs.update({
    'nodata': 9999,
    'transform': out_transform_mask_crop,
    'width': out_image_mask_crop.shape[2],
    'height': out_image_mask_crop.shape[1]
})
new_dataset = rasterio.open(
    'output/srtm_masked_cropped.tif',
    'w',
    **dst_kwargs
)
new_dataset.write(out_image_mask_crop)
new_dataset.close()
```

Let's also create a file connection to the newly created file `srtm_masked_cropped.tif` in order to plot it (Figure 5.2 (d)).

```
src_srtm_mask_crop = rasterio.open('output/srtm_masked_cropped.tif')
```

Figure 5.2 shows the original raster, and the three masking and/or cropping results.

```
# Original
fig, ax = plt.subplots(figsize=(3.5, 3.5))
rasterio.plot.show(src_srtm, ax=ax)
zion.plot(ax=ax, color='none', edgecolor='black');
# Masked
fig, ax = plt.subplots(figsize=(3.5, 3.5))
rasterio.plot.show(src_srtm_mask, ax=ax)
zion.plot(ax=ax, color='none', edgecolor='black');
# Cropped
fig, ax = plt.subplots(figsize=(3.5, 3.5))
rasterio.plot.show(out_image_crop, transform=out_transform_crop, ax=ax)
zion.plot(ax=ax, color='none', edgecolor='black');
# Masked+Cropped
fig, ax = plt.subplots(figsize=(3.5, 3.5))
rasterio.plot.show(src_srtm_mask_crop, ax=ax)
zion.plot(ax=ax, color='none', edgecolor='black');
```



(a) Original (b) Masked

(c) Cropped (d) Masked+Cropped

Figure 5.2: Raster masking and cropping

5.3 Raster extraction

Raster extraction is the process of identifying and returning the values associated with a 'target' raster at specific locations, based on a (typically vector) geographic 'selector' object. The reverse of raster extraction—assigning raster cell values based on vector objects—is rasterization, described in Section 5.4.

In the following examples, we use a package called **rasterstats**, which is specifically aimed at extracting raster values:

- To *points* (Section 5.3.1) or to *lines* (Section 5.3.2), via the `rasterstats.point_query` function
- To *polygons* (Section 5.3.3), via the `rasterstats.zonal_stats` function

5.3.1 Extraction to points

The simplest type of raster extraction is getting the values of raster cells at specific points. To demonstrate extraction to points, we will use `zion_points`, which contains a sample of 30 locations within the Zion National Park (Figure 5.3).

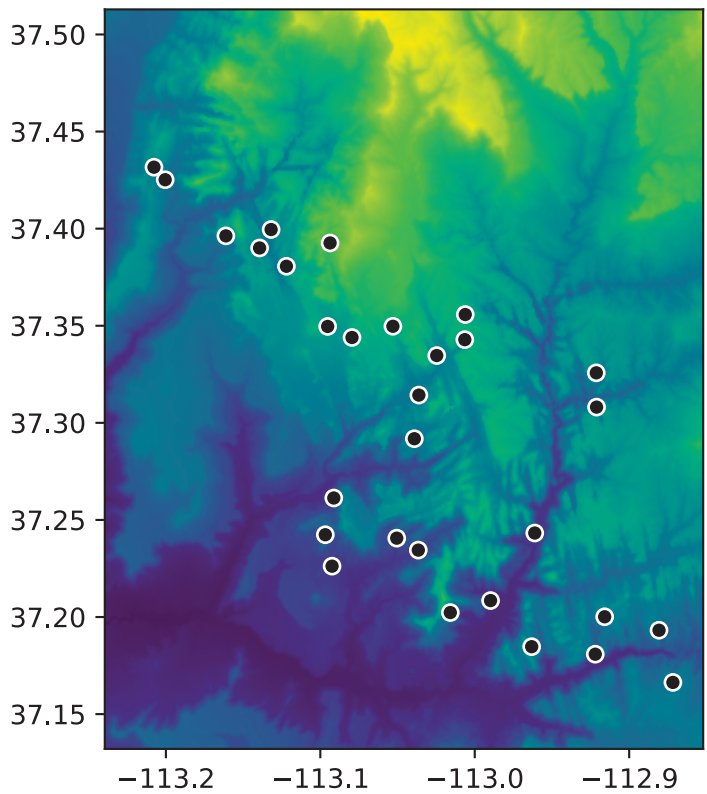


Figure 5.3: 30-point locations within the Zion National Park, with elevation in the background

```
fig, ax = plt.subplots()
rasterio.plot.show(src_srtm, ax=ax)
zion_points.plot(ax=ax, color='black', edgecolor='white');
```

The following expression extracts elevation values from `srtm.tif` according to `zion_points`, using `rasterstats.point_query`.

```
result1 = rasterstats.point_query(
    zion_points,
    src_srtm.read(1),
    nodata = src_srtm.nodata,
    affine = src_srtm.transform,
    interpolate='nearest'
)
```

The first two arguments are the vector layer and the array with raster values. The `nodata` and `affine` arguments are used to align the array values into the CRS and to correctly treat 'No Data' flags. Finally, the `interpolate` argument controls the way that the cell values are assigned to the point; `interpolate='nearest'` typically makes more sense, as opposed to the other option `interpolate='bilinear'` which is the default.

Alternatively, we can pass a raster file path to `rasterstats.point_query`, in which case `nodata` and `affine` are not necessary, as the function can understand those properties directly from the raster file.

```
result2 = rasterstats.point_query(
    zion_points,
    'data/srtm.tif',
    interpolate='nearest'
)
```

Either way, the resulting object is a `list` of raster values, corresponding to `zion_points`. For example, here are the elevations of the first five points.

```
result1[:5]
```

```
[1802, 2433, 1886, 1370, 1452]
```

To get a `GeoDataFrame` with the original points geometries (and other attributes, if any), as well as the extracted raster values, we can assign the extraction result into a new column. As you can see, both approaches give the same result.

```
zion_points['elev1'] = result1
zion_points['elev2'] = result2
zion_points
```

	geometry	elev1	elev2
0	POINT (-112.91587 37.20013)	1802	1802
1	POINT (-113.09369 37.39263)	2433	2433
2	POINT (-113.02462 37.33466)	1886	1886
...	...	...	...
27	POINT (-113.03655 37.23446)	1372	1372
28	POINT (-113.13933 37.39004)	1905	1905
29	POINT (-113.09677 37.24237)	1574	1574

The function supports extracting from just one raster band at a time. When passing an array, we can read the required band (as in, `.read(1)`, `.read(2)`, etc.). When passing a raster file path, we can set the band using the `band_num` argument (the default being `band_num=1`).

5.3.2 Extraction to lines

Raster extraction is also applicable with line selectors. The typical line extraction algorithm is to extract one value for each raster cell touched by a line. However, this particular approach is not recommended to obtain values along the transects, as it is hard to get the correct distance between each pair of extracted raster values.

For line extraction, a better approach is to split the line into many points (at equal distances along the line) and then extract the values for these points using the 'extraction to points' technique (Section 5.3.1). To demonstrate this, the code below creates (see Section 1.2 for recap) `zion_transect`, a straight line going from northwest to southeast of the Zion National Park.

```
coords = [[-113.2, 37.45], [-112.9, 37.2]]
zion_transect = shapely.LineString(coords)
print(zion_transect)
```

```
LINESTRING (-113.2 37.45, -112.9 37.2)
```

The utility of extracting heights from a linear selector is illustrated by imagining that you are planning a hike. The method demonstrated below provides an 'elevation profile' of the route (the line does not need to be straight), useful for estimating how long it will take due to long climbs.

First, we need to create a layer consisting of points along our line (`zion_transect`), at specified intervals (e.g., `250`). To do that, we need to transform the line into a projected CRS (so that we work with true distances, in m), such as UTM. This requires going through a `GeoSeries`, as **shapely** geometries have no CRS definition nor concept of reprojection (see Section 1.2.6).

```
zion_transect_utm = gpd.GeoSeries(zion_transect, crs=4326).to_crs(32612)
zion_transect_utm = zion_transect_utm.iloc[0]
```

The printout of the new geometry shows this is still a straight line between two points, only with coordinates in a projected CRS.

```
print(zion_transect_utm)
```

```
LINESTRING (305399.67208180577 4147066.650206682, 331380.8917453843 4118750.0947884847)
```

Next, we need to calculate the distances, along the line, where points are going to be generated. We do this using `np.arange`. The result is a numeric sequence starting at `0`, going up to line `.length`, in steps of `250` (m).

```
distances = np.arange(0, zion_transect_utm.length, 250)
distances[:7]  ## First 7 distance cutoff points
```

```
array([   0.,  250.,  500.,  750., 1000., 1250., 1500.])
```

The distance cutoffs are used to sample ('interpolate') points along the line. The **shapely** `.interpolate` method is used to generate the points, which then are reprojected back to the geographic CRS of the raster (EPSG:4326).

```
zion_transect_pnt = [zion_transect_utm.interpolate(d) for d in distances]
zion_transect_pnt = gpd.GeoSeries(zion_transect_pnt, crs=32612) \
    .to_crs(src_srtm.crs)
zion_transect_pnt
```

```
0              POINT (-113.2 37.45)
1       POINT (-113.19804 37.44838)
2       POINT (-113.19608 37.44675)
                    ...
151     POINT (-112.90529 37.20443)
152      POINT (-112.90334 37.2028)
153      POINT (-112.9014 37.20117)
Length: 154, dtype: geometry
```

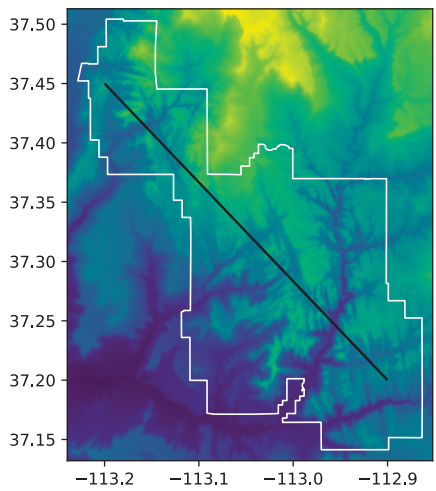
Finally, we extract the elevation values for each point in our transect and combine the information with `zion_transect_pnt` (after 'promoting' it to a `GeoDataFrame`, to accommodate extra attributes), using the point extraction method shown earlier (Section 5.3.1). We also attach the respective distance cutoff points `distances`.

```
result = rasterstats.point_query(
    zion_transect_pnt,
    src_srtm.read(1),
    nodata = src_srtm.nodata,
    affine = src_srtm.transform,
    interpolate='nearest'
)
zion_transect_pnt = gpd.GeoDataFrame(geometry=zion_transect_pnt)
zion_transect_pnt['dist'] = distances
zion_transect_pnt['elev'] = result
zion_transect_pnt
```

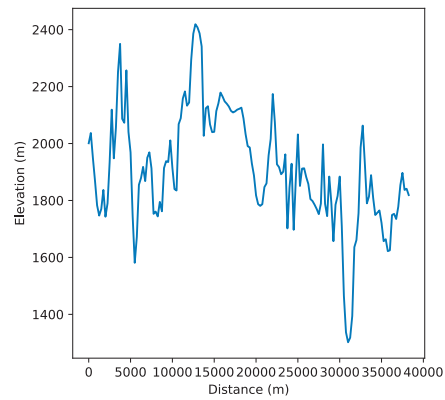
	geometry	dist	elev
0	POINT (-113.2 37.45)	0.0	2001
1	POINT (-113.19804 37.44838)	250.0	2037
2	POINT (-113.19608 37.44675)	500.0	1949
...	...	...	...
151	POINT (-112.90529 37.20443)	37750.0	1837
152	POINT (-112.90334 37.2028)	38000.0	1841
153	POINT (-112.9014 37.20117)	38250.0	1819

The information in `zion_transect_pnt`, namely the `'dist'` and `'elev'` attributes, can now be used to draw an elevation profile, as illustrated in Figure 5.4.

```
# Raster and a line transect
fig, ax = plt.subplots()
rasterio.plot.show(src_srtm, ax=ax)
gpd.GeoSeries(zion_transect).plot(ax=ax, color='black')
zion.plot(ax=ax, color='none', edgecolor='white');
# Elevation profile
fig, ax = plt.subplots()
zion_transect_pnt.set_index('dist')['elev'].plot(ax=ax)
ax.set_xlabel('Distance (m)')
ax.set_ylabel('Elevation (m)');
```



(a) Raster and a line transect



(b) Extracted elevation profile

Figure 5.4: Extracting a raster values profile to line

5.3.3 Extraction to polygons

The final type of geographic vector object for raster extraction is polygons. Like lines, polygons tend to return many raster values per vector geometry. For continuous rasters (Figure 5.5 (a)), we typically want to generate summary statistics for raster values per polygon, for example to characterize a single region or to compare many regions. The generation of raster summary statistics, by polygons, is demonstrated in the code below using `rasterstats.zonal_stats`, which creates a list of summary statistics (in this case a list of length 1, since there is just one polygon).

```
result = rasterstats.zonal_stats(
    zion,
    src_srtm.read(1),
    nodata = src_srtm.nodata,
    affine = src_srtm.transform,
    stats = ['mean', 'min', 'max']
)
result
```

```
[{'min': 1122.0, 'max': 2661.0, 'mean': 1818.211830154405}]
```

> **Note**
>
> `rasterstats.zonal_stats`, just like `rasterstats.point_query` (Section 5.3.1), supports raster input as file paths, rather than arrays plus `nodata` and `affine` arguments.

Transformation of the `list` to a `DataFrame` (e.g., to attach the derived attributes to the original polygon layer) is straightforward with the `pd.DataFrame` constructor.

```
pd.DataFrame(result)
```

	min	max	mean
0	1122.0	2661.0	1818.21183

Because there is only one polygon in the example, a `DataFrame` with a single row is returned. However, if `zion` was composed of more than one polygon, we would accordingly get more rows in the `DataFrame`. The result provides useful summaries, for example that the maximum height in the park is `2661` m above see level.

Note the `stats` argument, where we determine what type of statistics are calculated per polygon. Possible values other than `'mean'`, `'min'`, and `'max'` include:

- `'count'`—The number of valid (i.e., excluding 'No Data') pixels
- `'nodata'`—The number of pixels with 'No Data'

- `'majority'`—The most frequently occurring value
- `'median'`—The median value

See the documentation of `rasterstats.zonal_stats` for the complete list. Additionally, the `rasterstats.zonal_stats` function accepts user-defined functions for calculating any custom statistics.

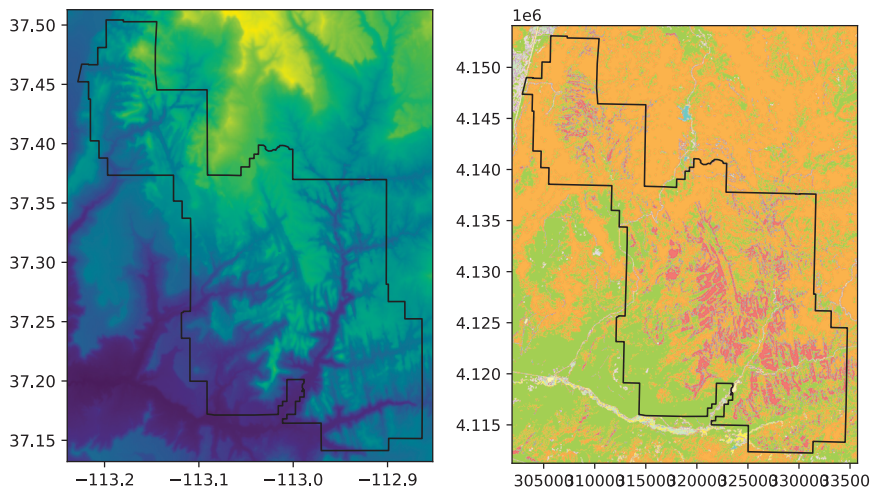
To count occurrences of categorical raster values within polygons (Figure 5.5 (b)), we can use masking (Section 5.2) combined with `np.unique`, as follows.

```
out_image, out_transform = rasterio.mask.mask(
    src_nlcd,
    zion.geometry.to_crs(src_nlcd.crs),
    crop=False,
    nodata=src_nlcd.nodata
)
counts = np.unique(out_image, return_counts=True)
counts
```

```
(array([  2,   3,   4,   5,   6,   7,   8, 255], dtype=uint8),
 array([  4205,  98285, 298299, 203701,    235,     62,    679, 852741]))
```

According to the result, for example, the value `2` ('Developed' class) appears in `4205` pixels within the Zion polygon.

Figure 5.5 illustrates the two types of raster extraction to polygons described above.



(a) Continuous raster (b) Categorical raster

Figure 5.5: Sample data used for continuous and categorical raster extraction to a polygon

```
# Continuous raster
fig, ax = plt.subplots()
rasterio.plot.show(src_srtm, ax=ax)
zion.plot(ax=ax, color='none', edgecolor='black');
# Categorical raster
fig, ax = plt.subplots()
rasterio.plot.show(src_nlcd, ax=ax, cmap='Set3')
zion.to_crs(src_nlcd.crs).plot(ax=ax, color='none', edgecolor='black');
```

5.4 Rasterization

Rasterization is the conversion of vector objects into their representation in raster objects. Usually, the output raster is used for quantitative analysis (e.g., analysis of terrain) or modeling. As we saw in Chapter 1, the raster data model has some characteristics that make it conducive to certain methods. Furthermore, the process of rasterization can help simplify datasets because the resulting values all have the same spatial resolution: rasterization can be seen as a special type of geographic data aggregation.

The **rasterio** package contains the `rasterio.features.rasterize` function for doing this work. To make it happen, we need to have the 'template' grid definition, i.e., the 'template' raster defining the extent, resolution and CRS of the output, in the `out_shape` (the output dimensions) and `transform` (the transformation matrix) arguments of `rasterio.features.rasterize`. In case we have an existing template raster, we simply need to query its `.shape` and `.transform`. On the other hand, if we need to create a custom template, e.g., covering the vector layer extent with specified resolution, there is some extra work to calculate both of these objects (see next example).

As for the vector geometries and their associated values, the `rasterio.features.rasterize` function requires the input vector shapes in the form of an iterable object of `geometry,value` pairs, where:

- `geometry` is the given geometry (**shapely** geometry object)
- `value` is the value to be 'burned' into pixels coinciding with the geometry (`int` or `float`)

Furthermore, we define how to deal with multiple values burned into the same pixel, using the `merge_alg` parameter. The default `merge_alg=rasterio.enums.MergeAlg.replace` means that 'later' values replace 'earlier' ones, i.e., the pixel gets the 'last' burned value. The other option `merge_alg=rasterio.enums.MergeAlg.add` means that burned values are summed, i.e., the pixel gets the sum of all burned values.

When rasterizing lines and polygons, we also have the choice between two pixel-matching algorithms. The default, `all_touched=False`, implies pixels that are selected by Bresenham's line algorithm[1] (for lines) or pixels whose center is within the polygon (for polygons). The other option `all_touched=True`, as the name suggests, implies that all pixels intersecting with the geometry are matched.

Finally, we can set the `fill` value, which is the value that 'unaffected' pixels get, with `fill=0` being the default.

How the `rasterio.features.rasterize` function works with all of these various parameters will be made clear in the next examples.

The geographic resolution of the 'template' raster has a major impact on the results: if it is too low (cell size is too large), the result may miss the full geographic variability of the vector data; if it is too high, computational times may be excessive. There are no simple rules to follow when deciding an appropriate geographic resolution, which is heavily dependent on the intended use of the results. Often the target resolution is imposed on the user, for example when the output of rasterization needs to be aligned to an existing raster.

Depending on the input data, rasterization typically takes one of two forms which we demonstrate next:

- in *point* rasterization (Section 5.4.1), we typically choose how to treat multiple points: either to summarize presence/absence, point count, or summed attribute values (Figure 5.6)
- in *line* and *polygon* rasterization (Section 5.4.2), there are typically no such 'overlaps' and we simply 'burn' attribute values, or fixed values, into pixels coinciding with the given geometries (Figure 5.7)

5.4.1 Rasterizing points

To demonstrate point rasterization, we will prepare a 'template' raster that has the same extent and CRS as the input vector data `cycle_hire_osm_projected` (a dataset on cycle hire points in London, illustrated in Figure 5.6 (a)) and a spatial resolution of 1000 m. To do that, we first take our point layer and transform it to a projected CRS.

```
cycle_hire_osm_projected = cycle_hire_osm.to_crs(27700)
```

Next, we calculate the `out_shape` and `transform` of the template raster. To calculate the transform, we combine the top-left corner of the `cycle_hire_osm_projected` bounding box with the required resolution (e.g., 1000 m).

[1] https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

```
bounds = cycle_hire_osm_projected.total_bounds
res = 1000
transform = rasterio.transform.from_origin(
    west=bounds[0],
    north=bounds[3],
    xsize=res,
    ysize=res
)
transform
```

```
Affine(1000.0, 0.0, np.float64(523038.61452275474),
       0.0, -1000.0, np.float64(184971.40854297992))
```

To calculate the `out_shape`, we divide the x-axis and y-axis extent by the resolution, taking the ceiling of the results.

```
rows = math.ceil((bounds[3] - bounds[1]) / res)
cols = math.ceil((bounds[2] - bounds[0]) / res)
shape = (rows, cols)
shape
```

```
(11, 16)
```

Finally, we are ready to rasterize. As mentioned above, point rasterization can be a very flexible operation: the results depend not only on the nature of the template raster, but also on the pixel 'activation' method, namely the way we deal with multiple points matching the same pixel.

To illustrate this flexibility, we will try three different approaches to point rasterization (Figure 5.6 (b)-(d)). First, we create a raster representing the presence or absence of cycle hire points (known as presence/absence rasters). In this case, we transfer the value of `1` to all pixels where at least one point falls in. In the **rasterio** framework, we use the `rasterio.features.rasterize` function, which requires an iterable object of `geometry,value` pairs. In this first example, we transform the point `GeoDataFrame` into a `list` of `shapely` geometries and the (fixed) value of `1`, using list comprehension, as follows. The first five elements of the `list` are hereby printed to illustrate its structure.

```
g = [(g, 1) for g in cycle_hire_osm_projected.geometry]
g[:5]
```

```
[(<POINT (532353.838 182857.655)>, 1),
 (<POINT (529848.35 183337.175)>, 1),
 (<POINT (530635.62 182608.992)>, 1),
 (<POINT (532540.398 182495.756)>, 1),
 (<POINT (530432.094 182906.846)>, 1)]
```

The list of `geometry,value` pairs is passed to `rasterio.features.rasterize`, along with the `out_shape` and `transform` which define the raster

template. The result `ch_raster1` is an `ndarray` with the burned values of `1` where the pixel coincides with at least one point, and `0` in 'unaffected' pixels. Note that `merge_alg=rasterio.enums.MergeAlg.replace` (the default) is used here, which means that a pixel gets `1` when one or more points fall in it, or keeps the original `0` value otherwise.

```
ch_raster1 = rasterio.features.rasterize(
    shapes=g,
    out_shape=shape,
    transform=transform
)
ch_raster1
```

```
array([[0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1],
       [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1],
       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
       [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
```

In our second variant of point rasterization, we count the number of bike hire stations. To do that, we use the fixed value of `1` (same as in the last example), but this time combined with the `merge_alg=rasterio.enums.MergeAlg.add` argument. That way, multiple values burned into the same pixel are *summed*, rather than replaced keeping last (which is the default). The new output, `ch_raster2`, shows the number of cycle hire points in each grid cell.

```
g = [(g, 1) for g in cycle_hire_osm_projected.geometry]
ch_raster2 = rasterio.features.rasterize(
    shapes=g,
    out_shape=shape,
    transform=transform,
    merge_alg=rasterio.enums.MergeAlg.add
)
ch_raster2
```

```
array([[ 0,  0,  0,  0,  0,  1,  1,  0,  0,  0,  0,  0,  0,  1,  3,  3],
       [ 0,  0,  0,  1,  3,  3,  5,  5,  8,  9,  1,  3,  2,  6,  7,  0],
       [ 0,  0,  0,  8,  5,  4, 11, 10, 12,  9, 11,  4,  8,  5,  4,  0],
       [ 0,  1,  4, 10, 10, 11, 18, 16, 13, 12,  8,  6,  5,  2,  3,  0],
       [ 3,  3,  9,  3,  5, 14, 10, 15,  9,  9,  5,  8,  0,  0, 12,  2],
       [ 4,  5,  9, 11,  6,  7,  7,  3, 10,  9,  4,  0,  0,  0,  0,  0],
       [ 4,  0,  7,  8,  8,  4, 11, 10,  7,  3,  0,  0,  0,  0,  0,  0],
       [ 0,  1,  3,  0,  0,  1,  4,  0,  1,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  1,  1,  0,  1,  0,  4,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  1,  0,  5,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]],
      dtype=uint8)
```

The cycle hire locations have different numbers of bicycles described by the capacity variable, raising the question, what is the capacity in each grid cell? To calculate that, in our third point rasterization variant we sum the field (`'capacity'`) rather than the fixed values of `1`. This requires using a more complex list comprehension expression, where we also (1) extract both geometries and the attribute of interest, and (2) filter out 'No Data' values, which can be done as follows. You are invited to run the separate parts to see how this works; the important point is that, in the end, we get the list `g` with the `geometry,value` pairs to be burned, only that the `value` is now variable, rather than fixed, among points.

```
g = [(g, v) for g, v in cycle_hire_osm_projected[['geometry', 'capacity']] \
        .dropna(subset='capacity')
        .to_numpy() \
        .tolist()]
g[:5]
```

```
[(<POINT (532353.838 182857.655)>, 14.0),
 (<POINT (530635.62 182608.992)>, 11.0),
 (<POINT (532620.775 181944.736)>, 20.0),
 (<POINT (527891.578 181374.392)>, 6.0),
 (<POINT (530399.064 181205.925)>, 17.0)]
```

Now we rasterize the points, again using `merge_alg=rasterio.enums.MergeAlg.add` to sum the capacity values per pixel.

```
ch_raster3 = rasterio.features.rasterize(
    shapes=g,
    out_shape=shape,
    transform=transform,
    merge_alg=rasterio.enums.MergeAlg.add
)
ch_raster3
```

```
array([[  0.,   0.,   0.,   0.,   0.,  11.,  34.,   0.,   0.,   0.,   0.,
          0.,   0.,  11.,  35.,  24.],
       [  0.,   0.,   0.,   7.,  30.,  46.,  60.,  73.,  72.,  75.,   6.,
         50.,  25.,  47.,  36.,   0.],
       [  0.,   0.,   0.,  89.,  36.,  31., 167.,  97., 115.,  80., 138.,
         61.,  65., 109.,  43.,   0.],
       [  0.,  11.,  42., 104., 108., 138., 259., 206., 203., 135., 107.,
         37.,   0.,  25.,  60.,   0.],
       [ 88.,  41.,  83.,  28.,  64., 115.,  99., 249., 107., 117.,  60.,
         33.,   0.,   0.,   0.,   0.],
       [  0.,  89., 107.,  95.,  73., 119.,  69.,  23., 140., 141.,  46.,
          0.,   0.,   0.,   0.,   0.],
       [  0.,   0.,  55.,  97., 101.,  59., 119., 109.,  75.,  12.,   0.,
          0.,   0.,   0.,   0.,   0.],
       [  0.,  10.,  23.,   0.,   0.,   5.,  41.,   0.,   8.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.],
```

```
[ 0.,  19.,   9.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
  0.,   0.,   0.,   0.,   0.],
[ 0.,  29.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
  0.,   0.,   0.,   0.,   0.],
[ 0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
  0.,   0.,   0.,   0.,   0.]], dtype=float32)
```

The result `ch_raster3` shows the total capacity of cycle hire points in each grid cell.

The input point layer `cycle_hire_osm_projected` and the three variants of rasterizing it `ch_raster1`, `ch_raster2`, and `ch_raster3` are shown in Figure 5.6.

```
# Input points
fig, ax = plt.subplots()
cycle_hire_osm_projected.plot(column='capacity', ax=ax);
# Presence/Absence
fig, ax = plt.subplots()
rasterio.plot.show(ch_raster1, transform=transform, ax=ax);
# Point counts
fig, ax = plt.subplots()
rasterio.plot.show(ch_raster2, transform=transform, ax=ax);
# Summed attribute values
fig, ax = plt.subplots()
rasterio.plot.show(ch_raster3, transform=transform, ax=ax);
```

5.4.2 Rasterizing lines and polygons

Another dataset based on California's polygons and borders (created below) illustrates rasterization of lines. There are three preliminary steps. First, we subset the California polygon.

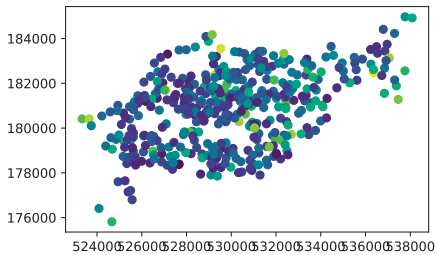
```
california = us_states[us_states['NAME'] == 'California']
california
```

	GEOID	NAME	...	total_pop_15	geometry
26	06	California	...	38421464.0	MULTIPOLYGON (((-118.60338 33.4...

Second, we 'cast' the polygon into a `'MultiLineString'` geometry, using the `.boundary` property that `GeoSeries` and `DataFrames` have.

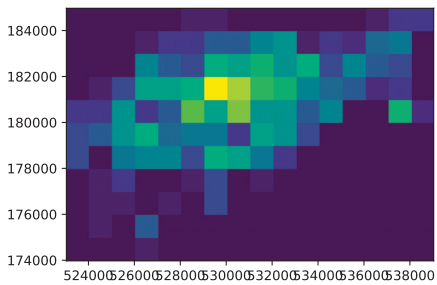
```
california_borders = california.boundary
california_borders
```

```
26    MULTILINESTRING ((-118.60338 33...
dtype: geometry
```

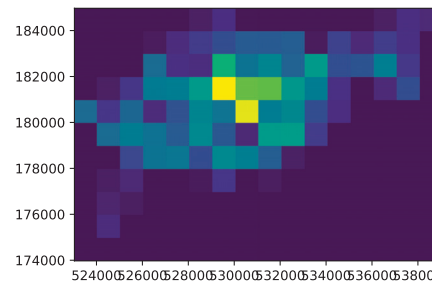


(a) Input points

(b) Presence/Absence



(c) Point counts



(d) Summed attribute values

Figure 5.6: Original data and three variants of point rasterization

Third, we create the `transform` and `shape` describing our template raster, with a resolution of `0.5` degree, using the same approach as in Section 5.4.1.

```
bounds = california_borders.total_bounds
res = 0.5
transform = rasterio.transform.from_origin(
    west=bounds[0],
    north=bounds[3],
    xsize=res,
    ysize=res
)
rows = math.ceil((bounds[3] - bounds[1]) / res)
cols = math.ceil((bounds[2] - bounds[0]) / res)
shape = (rows, cols)
shape
```

```
(19, 21)
```

Finally, we rasterize `california_borders` based on the calculated template's `shape` and `transform`. When considering line or polygon rasterization, one useful additional argument is `all_touched`. By default it is `False`, but when changed to `True`—all cells that are touched by a line or polygon border get a

value. Line rasterization with `all_touched=True` is demonstrated in the code below (Figure 5.7, left). We are also using `fill=np.nan` to set 'background' values to 'No Data'.

```
california_raster1 = rasterio.features.rasterize(
    [(g, 1) for g in california_borders],
    out_shape=shape,
    transform=transform,
    all_touched=True,
    fill=np.nan,
    dtype=np.float64
)
```

Compare it to polygon rasterization, with `all_touched=False` (the default), which selects only raster cells whose centroids are inside the selector polygon, as illustrated in Figure 5.7 (right).

```
california_raster2 = rasterio.features.rasterize(
    [(g, 1) for g in california.geometry],
    out_shape=shape,
    transform=transform,
    fill=np.nan,
    dtype=np.float64
)
```

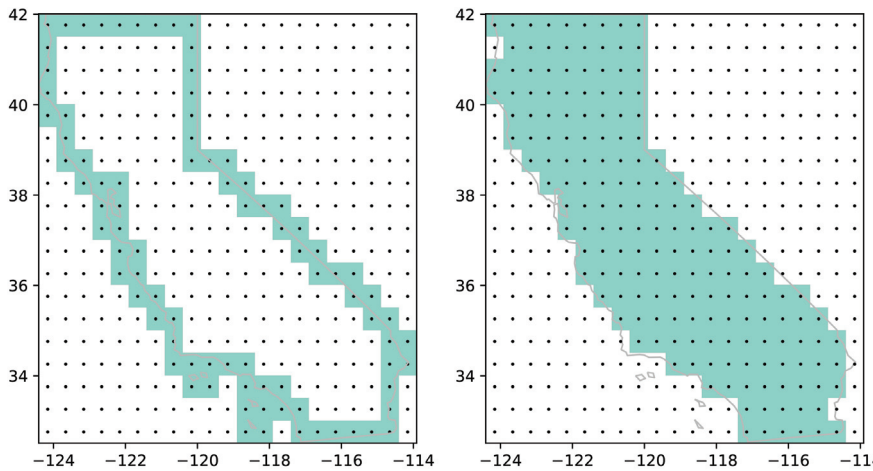
To illustrate which raster pixels are actually selected as part of rasterization, we also show them as points. This also requires the following code section to calculate the points, which we explain in Section 5.5.

```
height = california_raster1.shape[0]
width = california_raster1.shape[1]
cols, rows = np.meshgrid(np.arange(width), np.arange(height))
x, y = rasterio.transform.xy(transform, rows, cols)
x = np.array(x).flatten()
y = np.array(y).flatten()
z = california_raster1.flatten()
geom = gpd.points_from_xy(x, y, crs=california.crs)
pnt = gpd.GeoDataFrame(data={'value':z}, geometry=geom)
pnt
```

	value	geometry
0	1.0	POINT (-124.15959 41.75952)
1	1.0	POINT (-123.65959 41.75952)
2	1.0	POINT (-123.15959 41.75952)
...	...	...
396	1.0	POINT (-115.15959 32.75952)
397	1.0	POINT (-114.65959 32.75952)
398	NaN	POINT (-114.15959 32.75952)

Figure 5.7 shows the input vector layer, the rasterization results, and the points `pnt`.

```
# Line rasterization
fig, ax = plt.subplots()
rasterio.plot.show(california_raster1, transform=transform, ax=ax, cmap='Set3')
gpd.GeoSeries(california_borders).plot(ax=ax, edgecolor='darkgrey', linewidth=1)
pnt.plot(ax=ax, color='black', markersize=1);
# Polygon rasterization
fig, ax = plt.subplots()
rasterio.plot.show(california_raster2, transform=transform, ax=ax, cmap='Set3')
california.plot(ax=ax, color='none', edgecolor='darkgrey', linewidth=1)
pnt.plot(ax=ax, color='black', markersize=1);
```



(a) Line rasterization w/ `all_touched=True`

(b) Polygon rasterization w/ `all_touched=False`

Figure 5.7: Examples of line and polygon rasterization

5.5 Spatial vectorization

Spatial vectorization is the counterpart of rasterization (Section 5.4). It involves converting spatially continuous raster data into spatially discrete vector data such as points, lines, or polygons. There are three standard methods to convert a raster to a vector layer, which we cover next:

- Raster to polygons (Section 5.5.1)—converting raster cells to rectangular polygons, representing pixel areas

- Raster to points (Section 5.5.2)—converting raster cells to points, representing pixel centroids
- Raster to contours (Section 5.5.3)

Let us demonstrate all three in the given order.

5.5.1 Raster to polygons

The `rasterio.features.shapes` gives access to raster pixels as polygon geometries, along with the associated raster values. The returned object is a generator (see note in Section 3.3.1), yielding `geometry,value` pairs.

For example, the following expression returns a generator named `shapes`, referring to the pixel polygons.

```
shapes = rasterio.features.shapes(rasterio.band(src_grain, 1))
shapes
```

```
<generator object shapes at 0x7fdcfc1cb540>
```

We can generate all shapes at once into a `list` named `pol` with `list(shapes)`.

```
pol = list(shapes)
```

Each element in `pol` is a `tuple` of length 2, containing the GeoJSON-like `dict`—representing the polygon geometry and the value of the pixel(s) which comprise the polygon. For example, here is the first element of `pol`.

```
pol[0]
```

```
({'type': 'Polygon',
  'coordinates': [[(-1.5, 1.5),
    (-1.5, 1.0),
    (-1.0, 1.0),
    (-1.0, 1.5),
    (-1.5, 1.5)]]},
 1.0)
```

> **Note**
>
> Note that, when transforming a raster cell into a polygon, five-coordinate pairs need to be kept in memory to represent its geometry (explaining why rasters are often fast compared with vectors!).

To transform the `list` coming out of `rasterio.features.shapes` into the familiar `GeoDataFrame`, we need few more steps of data reshaping. First, we apply the `shapely.geometry.shape` function to go from a `list` of GeoJSON-like

dicts to a list of shapely geometry objects. The list can then be converted to a GeoSeries (see Section 1.2.6).

```
geom = [shapely.geometry.shape(i[0]) for i in pol]
geom = gpd.GeoSeries(geom, crs=src_grain.crs)
geom
```

```
0     POLYGON ((-1.5 1.5, -1.5 1, -1 ...
1     POLYGON ((-1 1.5, -1 1, -0.5 1,...
2     POLYGON ((-0.5 1.5, -0.5 1, 0 1...
                    ...
11    POLYGON ((0 -0.5, 0 -1, -0.5 -1...
12    POLYGON ((0.5 -1, 0.5 -1.5, 1 -...
13    POLYGON ((1 -1, 1 -1.5, 1.5 -1....
Length: 14, dtype: geometry
```

The values can also be extracted from the rasterio.features.shapes result and turned into a corresponding Series.

```
values = [i[1] for i in pol]
values = pd.Series(values)
values
```

```
0     1.0
1     0.0
2     1.0
     ...
11    2.0
12    0.0
13    2.0
Length: 14, dtype: float64
```

Finally, the two can be combined into a GeoDataFrame, hereby named result.

```
result = gpd.GeoDataFrame({'value': values, 'geometry': geom})
result
```

	value	geometry
0	1.0	POLYGON ((-1.5 1.5, -1.5 1, -1 ...
1	0.0	POLYGON ((-1 1.5, -1 1, -0.5 1,...
2	1.0	POLYGON ((-0.5 1.5, -0.5 1, 0 1...
...	...	...
11	2.0	POLYGON ((0 -0.5, 0 -1, -0.5 -1...
12	0.0	POLYGON ((0.5 -1, 0.5 -1.5, 1 -...
13	2.0	POLYGON ((1 -1, 1 -1.5, 1.5 -1....

The polygon layer result is shown in Figure 5.8.

```
result.plot(column='value', edgecolor='black', legend=True);
```

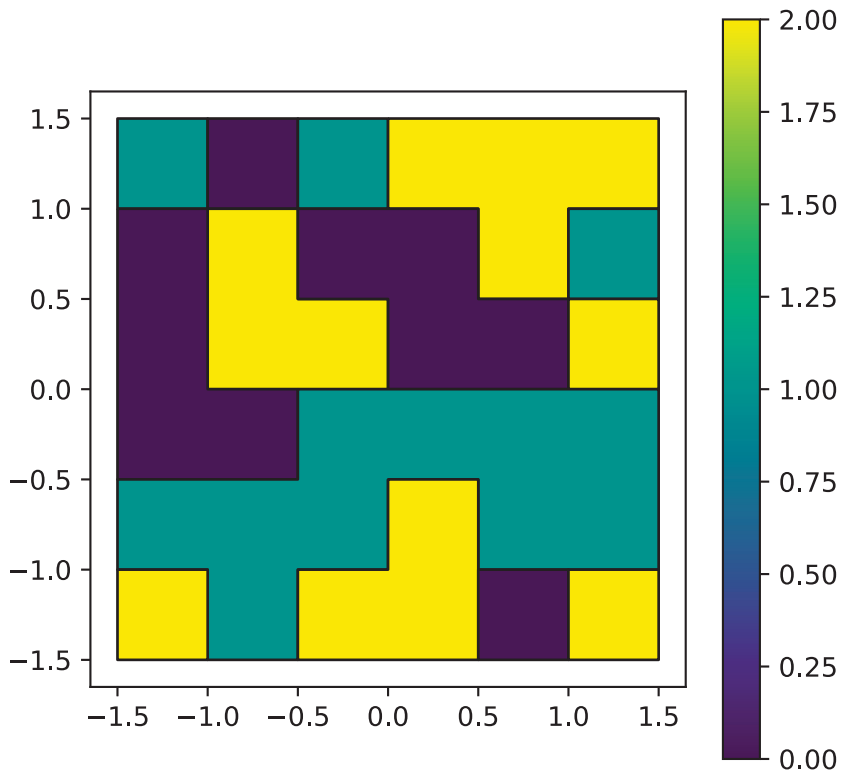


Figure 5.8: `grain.tif` converted to a polygon layer

As highlighted using `edgecolor='black'`, neighboring pixels sharing the same raster value are dissolved into larger polygons. The `rasterio.features.shapes` function unfortunately does not offer a way to avoid this type of dissolving. One suggestion is to add unique values between `0` and `0.9999` to all pixels, convert to polygons, and then get back to the original values using `np.floor`.

5.5.2 Raster to points

To transform a raster to points, we can use the `rasterio.transform.xy` function. As the name suggests, the function accepts row and column indices, and transforms them into x- and y-coordinates (using the raster's transformation matrix). For example, the coordinates of the top-left pixel can be calculated passing the `(row,col)` indices of `(0,0)`.

```
src = rasterio.open('output/elev.tif')
rasterio.transform.xy(src.transform, 0, 0)
```

```
(np.float64(-1.25), np.float64(1.25))
```

> **Note**
>
> Keep in mind that the coordinates of the top-left pixel (`(-1.25, 1.25)`), as calculated in the above expression, refer to the pixel *centroid*. Therefore, they are not identical to the raster origin coordinates (`(-1.5,1.5)`), as specified in the transformation matrix, which are the coordinates of the top-left edge/corner of the raster (see Figure 5.9).
>
> ```
> src.transform
> ```
>
> ```
> Affine(0.5, 0.0, -1.5,
>        0.0, -0.5, 1.5)
> ```

To generalize the above expression to calculate the coordinates of *all* pixels, we first need to generate a grid of all possible row/column index combinations. This can be done using `np.meshgrid`, as follows.

```
height = src.shape[0]
width = src.shape[1]
cols, rows = np.meshgrid(np.arange(width), np.arange(height))
```

We now have two arrays, `rows` and `cols`, matching the shape of `elev.tif` and containing the corresponding row and column indices.

```
rows
```

```
array([[0, 0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4, 4],
       [5, 5, 5, 5, 5, 5]])
```

```
cols
```

```
array([[0, 1, 2, 3, 4, 5],
       [0, 1, 2, 3, 4, 5],
       [0, 1, 2, 3, 4, 5],
       [0, 1, 2, 3, 4, 5],
       [0, 1, 2, 3, 4, 5],
       [0, 1, 2, 3, 4, 5]])
```

These can be passed to `rasterio.transform.xy` to transform the indices into point coordinates, accordingly stored in lists of arrays `x` and `y`.

```
x, y = rasterio.transform.xy(src.transform, rows, cols)
```

```
x
```

```
[array([-1.25, -0.75, -0.25,  0.25,  0.75,  1.25]),
 array([-1.25, -0.75, -0.25,  0.25,  0.75,  1.25]),
 array([-1.25, -0.75, -0.25,  0.25,  0.75,  1.25]),
 array([-1.25, -0.75, -0.25,  0.25,  0.75,  1.25]),
 array([-1.25, -0.75, -0.25,  0.25,  0.75,  1.25]),
 array([-1.25, -0.75, -0.25,  0.25,  0.75,  1.25])]
```

```
y
```

```
[array([1.25, 1.25, 1.25, 1.25, 1.25, 1.25]),
 array([0.75, 0.75, 0.75, 0.75, 0.75, 0.75]),
 array([0.25, 0.25, 0.25, 0.25, 0.25, 0.25]),
 array([-0.25, -0.25, -0.25, -0.25, -0.25, -0.25]),
 array([-0.75, -0.75, -0.75, -0.75, -0.75, -0.75]),
 array([-1.25, -1.25, -1.25, -1.25, -1.25, -1.25])]
```

Typically we want to work with the points in the form of a `GeoDataFrame` which also holds the attribute(s) value(s) as point attributes. To get there, we can transform the coordinates as well as any attributes to 1-dimensional arrays, and then use methods we are already familiar with (Section 1.2.6) to combine them into a `GeoDataFrame`.

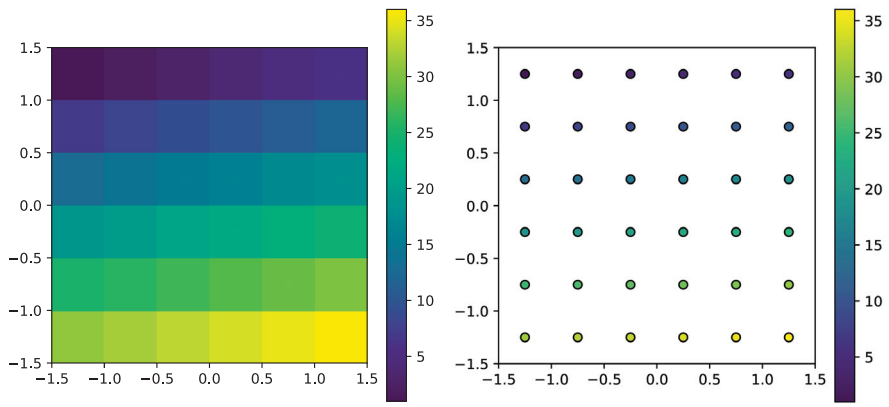
```
x = np.array(x).flatten()
y = np.array(y).flatten()
z = src.read(1).flatten()
geom = gpd.points_from_xy(x, y, crs=src.crs)
pnt = gpd.GeoDataFrame(data={'value':z}, geometry=geom)
pnt
```

	value	geometry
0	1	POINT (-1.25 1.25)
1	2	POINT (-0.75 1.25)
2	3	POINT (-0.25 1.25)
...	...	...
33	34	POINT (0.25 -1.25)
34	35	POINT (0.75 -1.25)
35	36	POINT (1.25 -1.25)

This 'high-level' workflow, like many other **rasterio**-based workflows covered in the book, is a commonly used one but lacking from the package itself. From the user's perspective, it may be a good idea to wrap the workflow into a function (e.g., `raster_to_points(src)`, returning a `GeoDataFrame`), to be re-used whenever we need it.

Figure 5.9 shows the input raster and the resulting point layer.

```
# Input raster
fig, ax = plt.subplots()
pnt.plot(column='value', legend=True, ax=ax)
rasterio.plot.show(src_elev, ax=ax);
# Points
fig, ax = plt.subplots()
pnt.plot(column='value', legend=True, edgecolor='black', ax=ax)
rasterio.plot.show(src_elev, alpha=0, ax=ax);
```



(a) Input raster (b) Points

Figure 5.9: Raster and point representation of `elev.tif`

Note that 'No Data' pixels can be filtered out from the conversion, if necessary (see Section 5.6).

5.5.3 Raster to contours

Another common type of spatial vectorization is the creation of contour lines, representing lines of continuous height or temperatures (*isotherms*), for example. We will use a real-world digital elevation model (DEM) because the artificial raster `elev.tif` produces parallel lines (task for the reader: verify this and explain why this happens). *Plotting* contour lines is straightforward, using the `contour=True` option of `rasterio.plot.show` (Figure 5.10).

```
fig, ax = plt.subplots()
rasterio.plot.show(src_dem, ax=ax)
rasterio.plot.show(
    src_dem,
    ax=ax,
    contour=True,
    levels=np.arange(0,1200,50),
    colors='black'
);
```

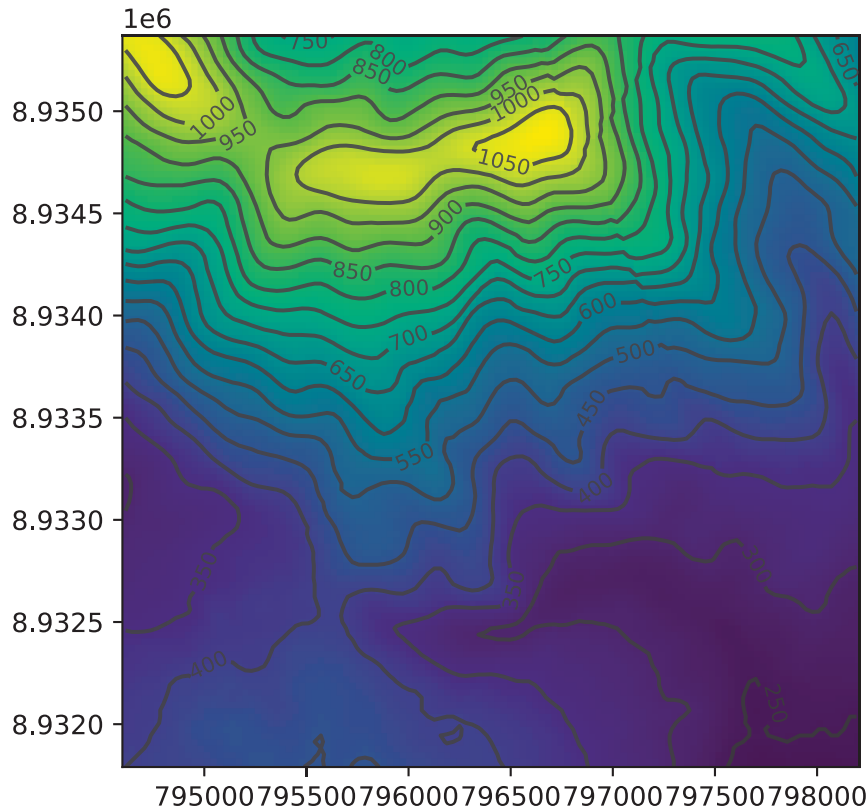


Figure 5.10: Displaying raster contours

Unfortunately, **rasterio** does not provide any way of extracting the contour lines in the form of a vector layer, for uses other than plotting.

There are two possible workarounds:

1. Using `gdal_contour` on the command line (see below), or through its Python interface **osgeo**
2. Writing a custom function to export contour coordinates generated by, e.g., **matplotlib** or **skimage**

We demonstrate the first approach, using `gdal_contour`. Although we deviate from the Python-focused approach towards more direct interaction with GDAL, the benefit of `gdal_contour` is the proven algorithm, customized to spatial data, and with many relevant options. Both the `gdal_contour` program (along with other GDAL programs) and its **osgeo** Python wrapper, should already be installed on your system since GDAL is a dependency of **rasterio**. Using the command line pathway, generating 50 m contours of the `dem.tif` file can be done as follows.

```
os.system('gdal_contour -a elev data/dem.tif output/dem_contour.gpkg -i 50.0')
```

Like all GDAL programs (also see `gdaldem` example in Section 3.3.4), `gdal_contour` works with files. Here, the input is the `data/dem.tif` file and the result is exported to the `output/dem_contour.gpkg` file.

To illustrate the result, let's read the resulting `dem_contour.gpkg` layer back into the Python environment. Note that the layer contains an attribute named `'elev'` (as specified using `-a elev`) with the contour elevation values.

```
contours1 = gpd.read_file('output/dem_contour.gpkg')
contours1
```

	ID	elev	geometry
0	0	750.0	LINESTRING (795382.355 8935384....
1	1	800.0	LINESTRING (795237.703 8935384....
2	2	650.0	LINESTRING (798098.379 8935384....
...	...	...	...
29	29	450.0	LINESTRING (795324.083 8931774....
30	30	450.0	LINESTRING (795488.616 8931774....
31	31	450.0	LINESTRING (795717.42 8931774.8...

Figure 5.11 shows the input raster and the resulting contour layer.

```
fig, ax = plt.subplots()
rasterio.plot.show(src_dem, ax=ax)
contours1.plot(ax=ax, edgecolor='black');
```

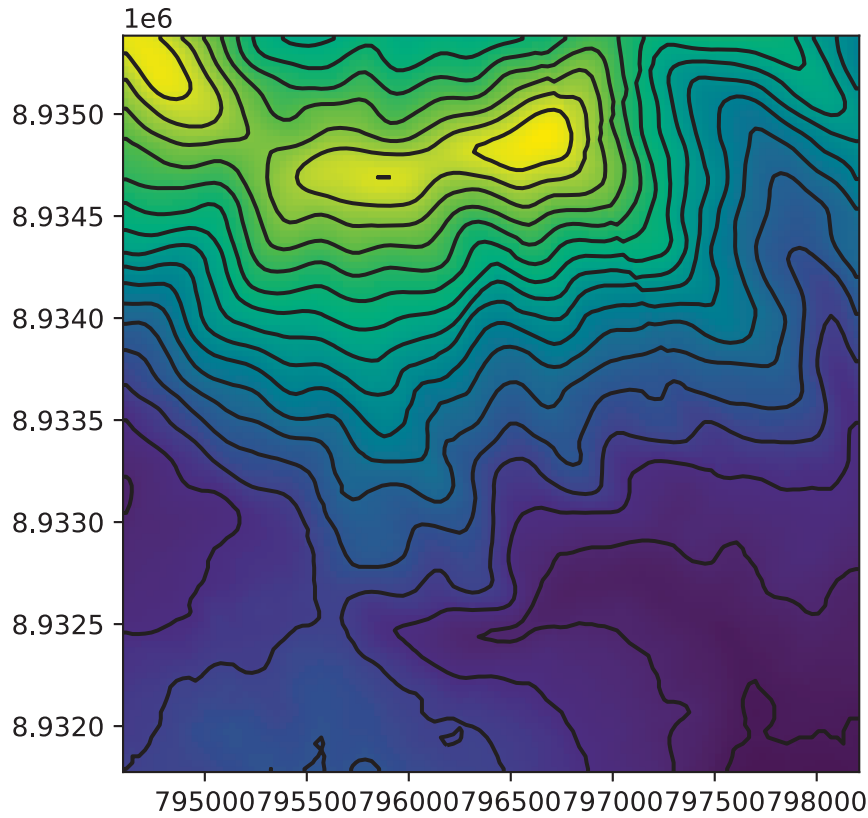


Figure 5.11: Contours of the `dem.tif` raster, calculated using the `gdal_contour` program

5.6 Distance to nearest geometry

Calculating a raster of distances to the nearest geometry is an example of a 'global' raster operation (Section 3.3.6). To demonstrate it, suppose that we need to calculate a raster representing the distance to the nearest coast in New Zealand. This example also wraps many of the concepts introduced in this chapter and in previous chapters, such as raster aggregation (Section 4.3.2), raster conversion to points (Section 5.5.2), and rasterizing points (Section 5.4.1).

For the coastline, we will dissolve the New Zealand administrative division polygon layer and 'extract' the boundary as a `'MultiLineString'` geometry (Figure 5.12). Note that `.dissolve(by=None)` (Section 2.2.2) calls `.union_all`

on all geometries (i.e., aggregates everything into one group), which is what we want to do here.

```
coastline = nz.dissolve().to_crs(src_nz_elev.crs).boundary.iloc[0]
coastline
```

Figure 5.12: New Zealand coastline geometry

For a 'template' raster, we will aggregate the New Zealand DEM, in the `nz_elev.tif` file, to 5 times coarser resolution. The code section below follows the aggregation example in Section 4.3.2.

```
factor = 0.2
# Reading aggregated array
r = src_nz_elev.read(1,
    out_shape=(
        int(src_nz_elev.height * factor),
        int(src_nz_elev.width * factor)
        ),
    resampling=rasterio.enums.Resampling.average
)
# Updating the transform
new_transform = src_nz_elev.transform * src_nz_elev.transform.scale(
    (src_nz_elev.width / r.shape[1]),
    (src_nz_elev.height / r.shape[0])
)
```

The resulting array `r/new_transform` and the lines layer `coastline` are plotted in Figure 5.13. Note that the raster values are average elevations based on 5×5 pixels, but this is irrelevant for the subsequent calculation; the raster is going to be used as a template, and all of its values will be replaced with distances to coastline (Figure 5.14).

```
fig, ax = plt.subplots()
rasterio.plot.show(r, transform=new_transform, ax=ax)
gpd.GeoSeries(coastline).plot(ax=ax, edgecolor='red');
```

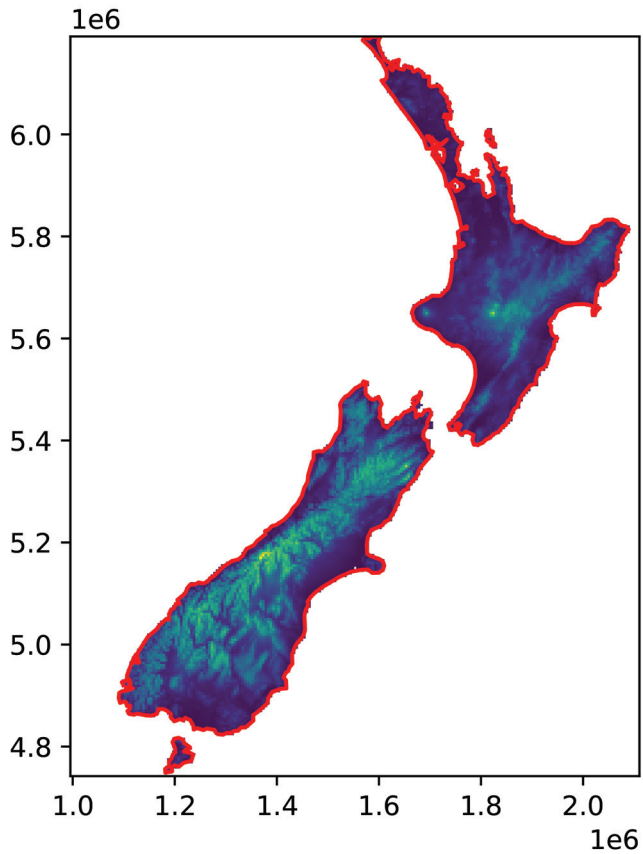


Figure 5.13: Template to calculate distance to nearest geometry (coastlines, in red)

To calculate the actual distances, we must convert each pixel to a vector (point) geometry. For this purpose, we use the technique demonstrated in Section 5.5.2, but we're keeping the points as a `list` of `shapely` geometries, rather than a `GeoDataFrame`, since such a list is sufficient for the subsequent calculation.

```
height = r.shape[0]
width = r.shape[1]
cols, rows = np.meshgrid(np.arange(width), np.arange(height))
x, y = rasterio.transform.xy(new_transform, rows, cols)
x = np.array(x).flatten()
y = np.array(y).flatten()
z = r.flatten()
x = x[~np.isnan(z)]
y = y[~np.isnan(z)]
geom = gpd.points_from_xy(x, y, crs=california.crs)
geom = list(geom)
geom[:5]
```

```
[<POINT (1572956.546 6189460.927)>,
 <POINT (1577956.546 6189460.927)>,
 <POINT (1582956.546 6189460.927)>,
 <POINT (1587956.546 6189460.927)>,
 <POINT (1592956.546 6189460.927)>]
```

The result `geom` is a `list` of `shapely` geometries, representing raster cell centroids (excluding `np.nan` pixels, which were filtered out).

Now we can calculate the corresponding `list` of point geometries and associated distances, using the `.distance` method from **shapely**:

```
distances = [(i, i.distance(coastline)) for i in geom]
distances[0]
```

```
(<POINT (1572956.546 6189460.927)>, 826.7523956221047)
```

Finally, we rasterize (see Section 5.4.1) the distances into our raster template.

```
image = rasterio.features.rasterize(
    distances,
    out_shape=r.shape,
    dtype=np.float64,
    transform=new_transform,
    fill=np.nan
)
image
```

```
array([[nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       ...,
       [nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan]])
```

The final result, a raster of distances to the nearest coastline, is shown in Figure 5.14.

```
fig, ax = plt.subplots()
rasterio.plot.show(image, transform=new_transform, ax=ax)
gpd.GeoSeries(coastline).plot(ax=ax, edgecolor='red');
```

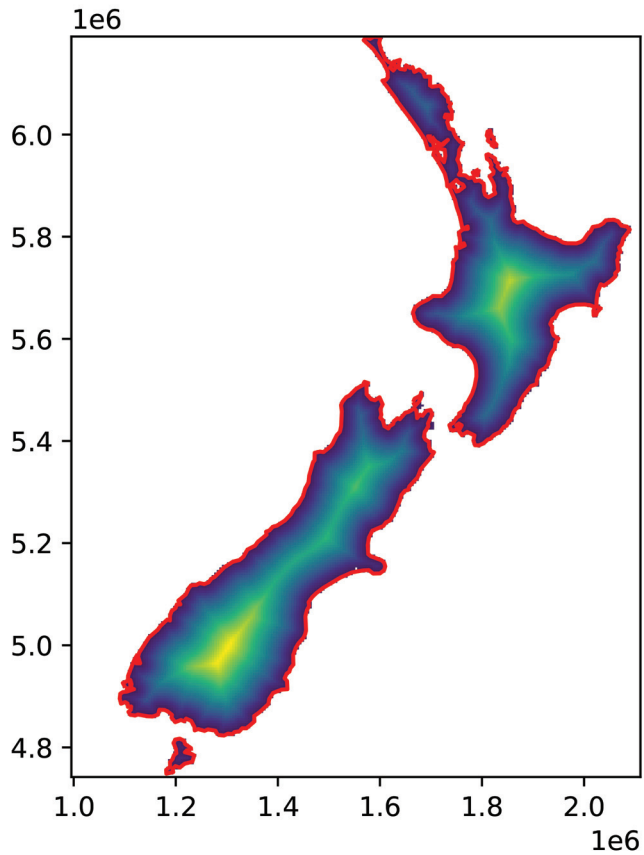


Figure 5.14: Distance to nearest coastline in New Zealand

6

Reprojecting geographic data

Prerequisites

This chapter requires importing the following packages:

```
import shutil
import math
import numpy as np
import matplotlib.pyplot as plt
import shapely
import pyproj
import geopandas as gpd
import rasterio
import rasterio.plot
import rasterio.warp
```

It also relies on the following data files:

```
src_srtm = rasterio.open('data/srtm.tif')
src_nlcd = rasterio.open('data/nlcd.tif')
zion = gpd.read_file('data/zion.gpkg')
world = gpd.read_file('data/world.gpkg')
cycle_hire_osm = gpd.read_file('data/cycle_hire_osm.gpkg')
```

6.1 Introduction

Section 1.4 introduced coordinate reference systems (CRSs), with a focus on the two major types: geographic ('lon/lat', with units in degrees longitude and latitude) and projected (typically with units of meters from a datum) coordinate systems. This chapter builds on that knowledge and goes further. It demonstrates how to set and transform geographic data from one CRS to another and, furthermore, highlights specific issues that can arise due to

DOI: 10.1201/9781003379911-6

ignoring CRSs that you should be aware of, especially if your data is stored with lon/lat coordinates.

It is important to know if your data is in a projected or geographic coordinate system, and the consequences of this for geometry operations. However, if you know the CRS of your data and the consequences for geometry operations (covered in the next section), CRSs should just work behind the scenes: people often suddenly need to learn about CRSs when things go wrong. Having a clearly defined project CRS that all project data is in, plus understanding how and why to use different CRSs, can ensure that things do not go wrong. Furthermore, learning about coordinate systems will deepen your knowledge of geographic datasets and how to use them effectively.

This chapter teaches the fundamentals of CRSs, demonstrates the consequences of using different CRSs (including what can go wrong), and how to 'reproject' datasets from one coordinate system to another. In the next section we introduce CRSs in Python, followed by Section 6.3 which shows how to get and set CRSs associated with spatial objects. Section 6.4 demonstrates the importance of knowing what CRS your data is in with reference to a worked example of creating buffers. We tackle questions of when to reproject and which CRS to use in Section 6.5 and Section 6.6, respectively. Finally, we cover reprojecting vector and raster objects in Section 6.7 and Section 6.8 and using custom projections in Section 6.9.

6.2 Coordinate Reference Systems

Most modern geographic tools that require CRS conversions, including Python packages and desktop GIS software such as QGIS, interface with PROJ, an open source C++ library that 'transforms coordinates from one coordinate reference system (CRS) to another'. CRSs can be described in many ways, including the following:

- Simple, yet potentially ambiguous, statements, such as 'it's in lon/lat coordinates'
- Formalized, yet now outdated, 'proj-strings', such as `+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs`
- With an identifying 'authority:code' text string, such as `EPSG:4326`

Each refers to the same thing: the 'WGS84' coordinate system that forms the basis of Global Positioning System (GPS) coordinates and many other datasets. But which one is correct?

The short answer is that the third way to identify CRSs is correct: `EPSG:4326` is understood by **geopandas** and **rasterio** packages covered in this book,

plus many other software projects for working with geographic data including QGIS and PROJ. `EPSG:4326` is future-proof. Furthermore, although it is machine readable, unlike the proj-string representation `EPSG:4326` is short, easy to remember and highly 'findable' online (searching for `EPSG:4326` yields a dedicated page on the website epsg.io[1], for example). The more concise identifier `4326` is also understood by **geopandas** and **rasterio**.

The longer answer is that none of the three descriptions is sufficient, and more detail is needed for unambiguous CRS handling and transformations: due to the complexity of CRSs, it is not possible to capture all relevant information about them in such short text strings. For this reason, the Open Geospatial Consortium (OGC, which also developed the Simple Features specification that the **geopandas** package implements) developed an open standard format for describing CRSs that is called WKT (Well Known Text). This is detailed in a 100+ page document that 'defines the structure and content of a text string implementation of the abstract model for coordinate reference systems described in ISO 19111:2019' (Open Geospatial Consortium 2019). The WKT representation of the WGS84 CRS, which has the identifier `EPSG:4326` is as follows.

```
crs = pyproj.CRS.from_string('EPSG:4326') # or '.from_epsg(4326)'
print(crs.to_wkt(pretty=True))
```

```
GEOGCRS["WGS 84",
    ENSEMBLE["World Geodetic System 1984 ensemble",
        MEMBER["World Geodetic System 1984 (Transit)"],
        MEMBER["World Geodetic System 1984 (G730)"],
        MEMBER["World Geodetic System 1984 (G873)"],
        MEMBER["World Geodetic System 1984 (G1150)"],
        MEMBER["World Geodetic System 1984 (G1674)"],
        MEMBER["World Geodetic System 1984 (G1762)"],
        MEMBER["World Geodetic System 1984 (G2139)"],
        ELLIPSOID["WGS 84",6378137,298.257223563,
            LENGTHUNIT["metre",1]],
        ENSEMBLEACCURACY[2.0]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["geodetic latitude (Lat)",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["geodetic longitude (Lon)",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
```

[1] https://epsg.io/4326

```
    USAGE[
        SCOPE["Horizontal component of 3D system."],
        AREA["World."],
        BBOX[-90,-180,90,180]],
    ID["EPSG",4326]]
```

The output of the command shows how the CRS identifier (also known as a Spatial Reference Identifier, or SRID) works: it is simply a look-up, providing a unique identifier associated with a more complete WKT representation of the CRS. This raises the question: what happens if there is a mismatch between the identifier and the longer WKT representation of a CRS? On this point Open Geospatial Consortium (Open Geospatial Consortium 2019) is clear, and the verbose WKT representation takes precedence over the identifier:

> Should any attributes or values given in the cited identifier be in conflict with attributes or values given explicitly in the WKT description, the WKT values shall prevail.

The convention of referring to CRSs identifiers in the form `AUTHORITY:CODE` allows a wide range of formally defined coordinate systems to be referred to. The most commonly used authority in CRS identifiers is EPSG, an acronym for the European Petroleum Survey Group which published a standardized list of CRSs. Other authorities can be used in CRS identifiers. `ESRI:54030`, for example, refers to ESRI's implementation of the Robinson projection, which has the following WKT string.

```
crs = pyproj.CRS.from_string('ESRI:54030')
print(crs.to_wkt(pretty=True))
```

```
PROJCRS["World_Robinson",
    BASEGEOGCRS["WGS 84",
        DATUM["World Geodetic System 1984",
            ELLIPSOID["WGS 84",6378137,298.257223563,
                LENGTHUNIT["metre",1]]],
        PRIMEM["Greenwich",0,
            ANGLEUNIT["Degree",0.0174532925199433]]],
    CONVERSION["World_Robinson",
        METHOD["Robinson"],
        PARAMETER["Longitude of natural origin",0,
            ANGLEUNIT["Degree",0.0174532925199433],
            ID["EPSG",8802]],
        PARAMETER["False easting",0,
            LENGTHUNIT["metre",1],
            ID["EPSG",8806]],
        PARAMETER["False northing",0,
            LENGTHUNIT["metre",1],
            ID["EPSG",8807]]],
```

```
    CS[Cartesian,2],
        AXIS["(E)",east,
            ORDER[1],
            LENGTHUNIT["metre",1]],
        AXIS["(N)",north,
            ORDER[2],
            LENGTHUNIT["metre",1]],
    USAGE[
        SCOPE["Not known."],
        AREA["World."],
        BBOX[-90,-180,90,180]],
    ID["ESRI",54030]]
```

WKT strings are exhaustive, detailed, and precise, allowing for unambiguous CRSs storage and transformations. They contain all relevant information about any given CRS, including its datum and ellipsoid, prime meridian, projection, and units.

Recent PROJ versions (6+) still allow use of proj-strings to define coordinate operations, but some proj-string keys (`+nadgrids`, `+towgs84`, `+k`, `+init=epsg:`) are either no longer supported or are discouraged. Additionally, only three datums (i.e., WGS84, NAD83, and NAD27) can be directly set in proj-string. Longer explanations of the evolution of CRS definitions and the PROJ library can be found in Bivand (2021), Chapter 2 of Pebesma and Bivand (2022), and a blog post by Floris Vanderhaeghe[2].

> **Note**
>
> As outlined in the PROJ documentation, there are different versions of the WKT CRS format including WKT1 and two variants of WKT2, the latter of which (WKT2, 2018 specification) corresponds to the ISO 19111:2019 (Open Geospatial Consortium 2019).

6.3 Querying and setting coordinate systems

Let's see how CRSs are stored in Python spatial objects and how they can be queried and set. First, we will look at getting and setting CRSs in vector geographic data objects. Consider the `GeoDataFrame` object named `world`, imported from a file `world.gpkg` that represents countries worldwide. Its CRS can be retrieved using the `.crs` property.

[2] https://inbo.github.io/tutorials/tutorials/spatial_crs_coding/

```
world.crs
```

```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

The output specifies the following pieces of information:

1. The CRS type (`Geographic 2D CRS`) and SRID code (`EPSG:4326`)
2. The CRS name (`WGS 84`)
3. The axes (`latitude`, `longitude`) and their units (`degree`)
4. The applicable area name (`World`) and bounding box (`(-180.0, -90.0, 180.0, 90.0)`)
5. The datum (`WGS 84`)

The WKT representation, which is internally used when saving the object to a file or doing any coordinate operations, can be extracted using `.crs.to_wkt()` as shown above (Section 6.2). We can also see that the `world` object has the WGS84 ellipsoid, the latitude and longitude axis order, and uses the Greenwich prime meridian. We also have the suitable area specification for the use of this CRS, and CRS identifier: `EPSG:4326`.

The CRS specification object, such as `world.crs`, has several useful properties and methods to explicitly retrieve information about the used CRS. For example, we can check whether the CRS is geographic with the `.is_geographic` property.

```
world.crs.is_geographic
```

```
True
```

CRS units of both axes (typically identical) can be retrieved with the `.axis_info` property.

```
world.crs.axis_info[0].unit_name, world.crs.axis_info[1].unit_name
```

```
('degree', 'degree')
```

`AUTHORITY` and `CODE` strings may be obtained with the `.to_authority()` method.

```
world.crs.to_authority()
```

```
('EPSG', '4326')
```

In cases when a coordinate reference system (CRS) is missing or the wrong CRS is set, the `.set_crs` method can be used on a `GeoSeries` or a `GeoDataFrame` to set it. The CRS can be specified using an EPSG code as the first argument. In case the object already has a different CRS definition, we must also specify `allow_override=True` to replace it (otherwise we get an error). In the first example we set the `EPSG:4326` CRS, which has no effect because `world` already has that exact CRS definition, while the second example replaces the existing CRS with a new definition of `EPSG:3857`.

```
world2 = world.set_crs(4326)
world3 = world.set_crs(3857, allow_override=True)
```

The provided number is interpreted as an `EPSG` code. We can also use strings, as in `'EPSG:4326'`, which is useful to make the code more clear and when using other authorities than `EPSG`.

```
world4 = world.set_crs('ESRI:54009', allow_override=True)
```

In **rasterio**, the CRS information is stored as part of a raster file connection metadata (Section 1.3.1). Replacing the CRS definition for a **rasterio** file connection is typically not necessary, because it is not considered in any operation; only the transformation matrix and coordinates are. One exception is when writing the raster, in which case we need to construct the metadata of the raster file to be written, and therein specify the CRS anyway (Section 1.3.2). However, if we, for some reason, need to change the CRS definition in the file connection metadata, we can do that when opening the file in `r+` (reading and writing) mode. To demonstrate, we will create a copy of the `nlcd.tif` file, named `nlcd_modified_crs.tif`,

```
shutil.copy('data/nlcd.tif', 'output/nlcd_modified_crs.tif')
```

```
'output/nlcd_modified_crs.tif'
```

and examine its existing CRS.

```
src_nlcd2 = rasterio.open('output/nlcd_modified_crs.tif', 'r+')
src_nlcd2.crs
```

```
CRS.from_epsg(26912)
```

> **Note**
>
> The `rasterio.open` function `mode`s generally follows Python's standard file connection modes, with possible arguments being `'r'` (read), `'w'` (write), `'r+'` (read/write), and `'w+'` (write/read) (the `'a'` 'append' mode is irrelevant for raster files). In the book, and in general, the most commonly used modes are `'r'` (read) and `'w'` (write). `'r+'`, used in the last example, means 'read/write'. Unlike with `'w'`, `'r+'` does not delete the existing content on open, making `'r+'` suitable for making changes in an existing file (such as here, replacing the CRS).

To replace the definition with a new one, such as `EPSG:3857`, we can use the `.crs` method, as shown below.

```
src_nlcd2.crs = 3857
src_nlcd2.close()
```

Next, examining the file connection demonstrates that the CRS was indeed changed.

```
rasterio.open('output/nlcd_modified_crs.tif').crs
```

```
CRS.from_epsg(3857)
```

Importantly, the `.set_crs` (for vector layers) or the assignment to `.crs` (for rasters), as shown above, do not alter coordinates' values or geometries. Their role is only to set a metadata information about the object CRS. Consequently, the objects we created, `world3`, `world4`, and `src_nlcd2` are 'incorrect', in the sense that the geometries are in fact given in a different CRS than specified in the associated CRS definition.

In some cases, the CRS of a geographic object is unknown, as is the case in the London dataset created in the code chunk below, building on the example of London introduced in Section 1.2.6.

```
lnd_point = shapely.Point(-0.1, 51.5)
lnd_geom = gpd.GeoSeries([lnd_point])
lnd_layer = gpd.GeoDataFrame({'geometry': lnd_geom})
lnd_layer
```

	geometry
0	POINT (-0.1 51.5)

Querying the `.crs` of such a layer returns `None`, therefore nothing is printed.

```
lnd_layer.crs
```

This implies that **geopandas** does not know what the CRS is and is unwilling to guess. Unless a CRS is manually specified or is loaded from a source that has CRS metadata, **geopandas** does not make any explicit assumptions about which coordinate systems, other than to say 'I don't know'. This behavior makes sense given the diversity of available CRSs but differs from some approaches, such as the GeoJSON file format specification, which makes the simplifying assumption that all coordinates have a lon/lat CRS: `EPSG:4326`.

A CRS can be added to `GeoSeries` or `GeoDataFrame` objects using the `.set_crs` method, as mentioned above.

```
lnd_layer = lnd_layer.set_crs(4326)
```

When working with **geopandas** and **rasterio**, datasets without a specified CRS are not an issue in most workflows, since only the coordinates are considered. It is up to the user to make sure that, when working with more than one layer, all of the coordinates are given in the same CRS (whether specified or not). When exporting the results, though, it is important to keep the CRS definition in place, because other software typically *do* use, and require, the CRS definition in calculations. It should also be mentioned that, in some cases the CRS specification is left unspecified on purpose, for example when working with layers in arbitrary or non-geographic space (simulations, internal building plans, analysis of plot-scale ecological patterns, etc.).

6.4 Geometry operations on projected and unprojected data

The **geopandas** package, through its dependency **shapely**, assumes planar geometry and works with distance/area values assumed to be in CRS units. In fact, the CRS definition is typically ignored, and the respective functions (such as in plotting and distance calculations) are applied on the 'bare' **shapely** geometries. Accordingly, it is crucial to make sure that:

- Geometric calculations are only applied in projected CRS
- If there is more than one layer involved—all layers have to be in the same (projected) CRS
- Distance and area values, are passed, and returned, in CRS units

For example, to calculate a buffer of 100 km around London, we need to work with a layer representing London in a projected CRS (e.g., `EPSG:27700`) and pass the distance value in the CRS units (e.g., `100000` m).

In the following code chunk we create, from scratch, a point layer `lnd_layer_proj` with a point representing London (compare it to `lnd_layer`, in a geographical CRS which we created above, see Section 6.3).

```
lnd_point_proj = shapely.Point(530000, 180000)
lnd_geom_proj = gpd.GeoSeries([lnd_point_proj], crs=27700)
lnd_layer_proj = gpd.GeoDataFrame({'geometry': lnd_geom_proj})
lnd_layer_proj
```

	geometry
0	POINT (530000 180000)

Now, we can use the `.buffer` method (Section 4.2.3) to calculate the buffer of 100 *km* around London.

```
lnd_layer_proj_buff = lnd_layer_proj.buffer(100000)
lnd_layer_proj_buff
```

```
0    POLYGON ((630000 180000, 629518...
dtype: geometry
```

The resulting buffer is shown in the left panel of Figure 6.1.

Calculating a 100-*km* buffer directly for `lnd_layer`, which is in a geographical CRS, is impossible. Since the `lnd_layer` is in decimal degrees, the closest thing to a 100-*km* buffer would be to use a distance of 1 degree, which is roughly equivalent to 100 *km* (1 degree is about 111 *km* at the equator):

```
lnd_layer_buff = lnd_layer.buffer(1)
lnd_layer_buff
```

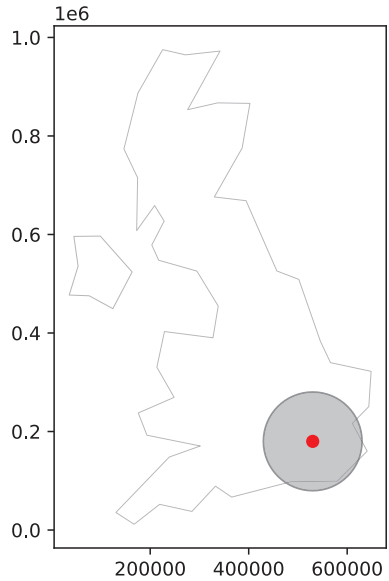
```
/tmp/ipykernel_151433/855451079.py:1: UserWarning:

Geometry is in a geographic CRS. Results from 'buffer' are
likely incorrect. Use 'GeoSeries.to_crs()' to re-project
geometries to a projected CRS before this operation.
```

```
0    POLYGON ((0.9 51.5, 0.89518 51....
dtype: geometry
```

However, this is incorrect, as told by the warning message and shown in the right panel of Figure 6.1. The association between degrees and true distance varies over the surface of the earth and we cannot assume it is fixed.

```
uk = world[world['name_long'] == 'United Kingdom']
uk_proj = uk.to_crs(27700)
# Around projected point
base = uk_proj.plot(color='none', edgecolor='darkgrey', linewidth=0.5)
lnd_layer_proj_buff.plot(color='grey', edgecolor='black', alpha=0.5, ax=base)
lnd_layer_proj.plot(color='red', ax=base);
# Around point in lon/lat
base = uk.plot(color='none', edgecolor='darkgrey', linewidth=0.5)
lnd_layer_buff.plot(color='grey', edgecolor='black', alpha=0.5, ax=base)
lnd_layer.plot(color='red', ax=base);
```



(a) Around a projected point and distance of 100 *km*

(b) Around a point in lon/lat using distance of 1 degree (incorrectly approximating 100 *km*)

Figure 6.1: Buffers around London

> **Note**
>
> The distance between two lines of longitude, called meridians, is around 111 *km* at the equator (execute `import geopy.distance;geopy.distance.geodesic((0,0),(0,1))` to find the precise distance). This shrinks to zero at the poles. At the latitude of London, for example, meridians are less than 70 *km* apart (challenge: execute code that verifies this). Lines of latitude, by contrast, are equidistant from each other irrespective of latitude: they are always around 111 *km* apart, including at the equator and near the poles.

> **Note**
>
> The **spherely**[3] package, in early stages of development at the time of writing, is aimed at providing a spherical-geometry counterpart to **shapely**, so that true distances (in m) and areas (in m^2) can be directly calculated on geometries in geographic CRS.

[3] https://github.com/benbovy/spherely

6.5 When to reproject?

The previous section showed how to set the CRS manually, with an expression such as `lnd_layer.set_crs(4326)`. In real-world applications, however, CRSs are usually set automatically when data is read-in. Thus, in many projects the main CRS-related task is to transform objects, from one CRS into another. But when should data be transformed? And into which CRS? There are no clear-cut answers to these questions and CRS selection always involves trade-offs (Maling 1992). However, there are some general principles provided in this section that can help you decide.

First, it's worth considering when to transform. In some cases, transformation to a geographic CRS is essential, such as when publishing data online (for example, a Leaflet-based map using Python package **folium**). Another case is when two objects with different CRSs must be compared or combined, as shown when we try to find the distance between two objects with different CRSs.

```
lnd_layer.distance(lnd_layer_proj)
```

```
/tmp/ipykernel_151433/2145313019.py:1: UserWarning:

Geometry is in a geographic CRS. Results from 'distance' are
likely incorrect. Use 'GeoSeries.to_crs()' to re-project
geometries to a projected CRS before this operation.


/tmp/ipykernel_151433/2145313019.py:1: UserWarning:

CRS mismatch between the CRS of left geometries and the CRS of
right geometries.
Use `to_crs()` to reproject one of the input geometries to match
the CRS of the other.

Left CRS: EPSG:4326
Right CRS: EPSG:27700


0    559715.614087
dtype: float64
```

Here, we got a meaningless distance value of `559715`, and a warning.

To make the `lnd_layer` and `lnd_layer_proj` objects geographically comparable, one of them must be transformed into the CRS of the other. But which CRS to use? The answer depends on context: many projects, especially those

involving web mapping, require outputs in `EPSG:4326`, in which case it is worth transforming the projected object. If, however, the project requires geometric calculations, implying planar geometry, e.g., calculating buffers (Section 6.4), it is necessary to transform data with a geographic CRS into an equivalent object with a projected CRS, such as the British National Grid (`EPSG:27700`). That is the subject of Section 6.6.

6.6 Which CRS to use?

The question of which CRS is tricky, and there is rarely a 'right' answer: 'There exist no all-purpose projections, all involve distortion when far from the center of the specified frame' (Bivand, Pebesma, and Gómez-Rubio 2013). Additionally, you should not be attached just to one projection for every task. It is possible to use one projection for some part of the analysis, another projection for a different part, and even some other for visualization. Always try to pick the CRS that serves your goal best!

When selecting *geographic* CRSs, the answer is often WGS84. It is used not only for web mapping, but also because GPS datasets and thousands of raster and vector datasets are provided in this CRS by default. WGS84 is the most common CRS in the world, so it is worth knowing its EPSG code: `4326`. This 'magic number' can be used to convert objects with unusual projected CRSs into something that is widely understood.

What about when a *projected* CRS is required? In some cases, it is not something that we are free to decide: 'often the choice of projection is made by a public mapping agency' (Bivand, Pebesma, and Gómez-Rubio 2013). This means that when working with local data sources, it is likely preferable to work with the CRS in which the data was provided, to ensure compatibility, even if the official CRS is not the most accurate. The example of London was easy to answer because the British National Grid (with its associated EPSG code `27700`) is well known, and the original dataset (`lnd_layer`) already had that CRS.

A commonly used default is Universal Transverse Mercator (UTM), a set of CRSs that divide the Earth into 60 longitudinal wedges and 20 latitudinal segments. The transverse Mercator projection used by UTM CRSs is conformal but distorts areas and distances with increasing severity with distance from the center of the UTM zone. Documentation from the GIS software Manifold therefore suggests restricting the longitudinal extent of projects using UTM zones to 6 degrees from the central meridian[4]. Therefore, we recommend using

[4] http://www.manifold.net/doc/mfd9/universal_transverse_mercator_projection.htm

UTM only when your focus is on preserving angles for a relatively small area!

Almost every place on Earth has a UTM code, such as `'60H'` which refers, among others, to northern New Zealand. UTM EPSG codes run sequentially from `32601` to `32660` for northern hemisphere locations and from `32701` to `32760` for southern hemisphere locations.

To show how the system works, let's create a function, `lonlat2UTM` to calculate the EPSG code associated with any point on the planet.

```
def lonlat2UTM(lon, lat):
    utm = (math.floor((lon + 180) / 6) % 60) + 1
    if lat > 0:
        utm += 32600
    else:
        utm += 32700
    return utm
```

The following command uses this function to identify the UTM zone and associated EPSG code for Auckland.

```
lonlat2UTM(174.7, -36.9)
```

```
32760
```

Here is another example for London (where we 'unpack' the coordinates of the 1st geometry in `lnd_layer` into the `lonlat2UTM` function arguments).

```
lonlat2UTM(*lnd_layer.geometry.iloc[0].coords[0])
```

```
32630
```

Currently, we also have tools helping us to select a proper CRS. For example, the webpage https://crs-explorer.proj.org/ lists CRSs based on selected location and type. Important note: while these tools are helpful in many situations, you need to be aware of the properties of the recommended CRS before you apply it.

In cases where an appropriate CRS is not immediately clear, the choice of CRS should depend on the properties that are most important to preserve in the subsequent maps and analysis. All CRSs are either equal-area, equidistant, conformal (with shapes remaining unchanged), or some combination of compromises of those (Section 1.4.2). Custom CRSs with local parameters can be created for a region of interest and multiple CRSs can be used in projects when no single CRS suits all tasks. 'Geodesic calculations' can provide a fall-back if no CRSs are appropriate[5]. Regardless of the projected CRS used, the results may not be accurate for geometries covering hundreds of kilometers.

[5] https://proj.org/geodesic.html

When deciding on a custom CRS, we recommend the following:

- A Lambert azimuthal equal-area (LAEA) projection for a custom local projection (set latitude and longitude of origin to the center of the study area), which is an equal-area projection at all locations but distorts shapes beyond thousands of kilometers
- Azimuthal equidistant (AEQD) projections for a specifically accurate straight-line distance between a point and the center point of the local projection
- Lambert conformal conic (LCC) projections for regions covering thousands of kilometers, with the cone set to keep distance and area properties reasonable between the secant lines
- Stereographic (STERE) projections for polar regions, but taking care not to rely on area and distance calculations thousands of kilometers from the center

One possible approach to automatically select a projected CRS specific to a local dataset is to create an azimuthal equidistant (AEQD) projection for the center-point of the study area. This involves creating a custom CRS (with no EPSG code) with units of meters based on the center point of a dataset. Note that this approach should be used with caution: no other datasets will be compatible with the custom CRS created and results may not be accurate when used on extensive datasets covering hundreds of kilometers.

The principles outlined in this section apply equally to vector and raster datasets. Some features of CRS transformation however are unique to each geographic data model. We will cover the particularities of vector data transformation in Section 6.7 and those of raster transformation in Section 6.8. The last section, Section 6.9, shows how to create custom map projections.

6.7 Reprojecting vector geometries

Section 1.2 demonstrated how vector geometries are made-up of points, and how points form the basis of more complex objects such as lines and polygons. Reprojecting vectors thus consists of transforming the coordinates of these points, which form the vertices of lines and polygons.

Section 6.4 contains an example in which at a `GeoDataFrame` had to be transformed into an equivalent object, with a different CRS, to calculate the distance between two objects. Reprojection of vector layers is done using the `.to_crs` method.

```
lnd_layer2 = lnd_layer.to_crs(27700)
```

Now that a transformed version of `lnd_layer` has been created, the distance between the two representations of London can be found using the `.distance` method.

```
lnd_layer2.distance(lnd_layer_proj)
```

```
0    2017.949587
dtype: float64
```

It may come as a surprise that `lnd_layer` and `lnd_layer2` are just over 2 *km* apart! The difference in location between the two points is not due to imperfections in the transforming operation (which is in fact very accurate) but the low precision of the manually specified coordinates when creating `lnd_layer` and `lnd_layer_proj`.

Reprojecting to a different CRS is also demonstrated below using `cycle_hire_osm`, a point layer that represents 'docking stations' where you can hire bicycles in London. The contents of the CRS object associated with a given geometry column are changed when the object's CRS is transformed. In the code chunk below, we create a new version of `cycle_hire_osm` with a projected CRS.

```
cycle_hire_osm_projected = cycle_hire_osm.to_crs(27700)
cycle_hire_osm_projected.crs
```

```
<Projected CRS: EPSG:27700>
Name: OSGB36 / British National Grid
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: United Kingdom (UK) - offshore to boundary of UKCS
within 49°45'N to 61°N and 9°W to 2°E; onshore Great Britain
(England, Wales and Scotland). Isle of Man onshore.
- bounds: (-9.01, 49.75, 2.01, 61.01)
Coordinate Operation:
- name: British National Grid
- method: Transverse Mercator
Datum: Ordnance Survey of Great Britain 1936
- Ellipsoid: Airy 1830
- Prime Meridian: Greenwich
```

The resulting object has a new CRS according to the EPSG code `27700`. How to find out more details about this EPSG code, or any code? One option is to search for it online. Another option is to create a standalone CRS object within the Python environment (using `pyproj.CRS.from_string` or `pyproj.CRS.from_epsg`, see Section 6.2), and then query its properties, such as `.name` and `.to_wkt()`.

```
crs_lnd_new = pyproj.CRS.from_epsg(27700)
crs_lnd_new.name, crs_lnd_new.to_wkt()
```

```
('OSGB36 / British National Grid',
 'PROJCRS["OSGB36 / British National Grid",BASEGEOGCRS["OSGB36",
 DATUM["Ordnance Survey of Great Britain 1936",ELLIPSOID
 ["Airy 1830",6377563.396,299.3249646,LENGTHUNIT["metre",1]]],
 PRIMEM["Greenwich",0,ANGLEUNIT["degree",0.0174532925199433]],
 ID["EPSG",4277]],CONVERSION["British National Grid",
 METHOD["Transverse Mercator",ID["EPSG",9807]],
 PARAMETER["Latitude of natural origin",49,
 ANGLEUNIT["degree",0.0174532925199433],ID["EPSG",8801]],
 PARAMETER["Longitude of natural origin",-2,ANGLEUNIT["degree",
 0.0174532925199433],ID["EPSG",8802]],PARAMETER["Scale
 factor at natural origin",0.9996012717,SCALEUNIT["unity",1],
 ID["EPSG",8805]],PARAMETER["False easting",400000,
 LENGTHUNIT["metre",1],ID["EPSG",8806]],
 PARAMETER["False northing",-100000,LENGTHUNIT["metre",1],
 ID["EPSG",8807]]],CS[Cartesian,2],
 AXIS["(E)",east,ORDER[1],LENGTHUNIT["metre",1]],
 AXIS["(N)",north,ORDER[2],LENGTHUNIT["metre",1]],
 USAGE[SCOPE["Engineering survey, topographic mapping."],
 AREA["United Kingdom (UK) - offshore to boundary of UKCS
 within 49°45\'N to 61°N and 9°W to 2°E; onshore Great Britain
 (England, Wales and Scotland). Isle of Man onshore."],
 BBOX[49.75,-9.01,61.01,2.01]],ID["EPSG",27700]]')
```

The result shows that the EPSG code 27700 represents the British National Grid, a result that could have been found by searching online for 'EPSG 27700'.

6.8 Reprojecting raster geometries

The CRSs concepts described in the previous section apply equally to rasters. However, there are important differences in reprojection of vectors and rasters: transforming a vector object involves changing the coordinates of every vertex, but this does not apply to raster data. Rasters are composed of rectangular cells of the same size (expressed by map units, such as degrees or meters), so it is usually impracticable to transform coordinates of pixels separately. Raster reprojection involves creating a new raster object in the destination CRS, often with a different number of columns and rows than the original. The attributes must subsequently be re-estimated, allowing the new pixels to be 'filled' with appropriate values. In other words, raster reprojection can

be thought of as two separate spatial operations: a vector reprojection of the raster extent to another CRS (Section 6.7), and computation of new pixel values through resampling (Section 4.3.3). Due to this additional complexity, in most cases when both raster and vector data are used, it is better to avoid reprojecting rasters and reproject vectors instead.

> **Note**
>
> Reprojection of regular rasters is also known as warping. Additionally, there is a second similar operation called 'transformation'. Instead of resampling all of the values, it leaves all values intact but recomputes new coordinates for every raster cell, changing the grid geometry. For example, it could convert the input raster (a regular grid) into a curvilinear grid. The **rasterio**, like common raster file formats (such as GeoTIFF), does not support curvilinear grids. The **xarray** package, for instance, can be used to work with curvilinear grids.

The raster reprojection process is done using two functions from the `rasterio.warp` sub-package:

1. `rasterio.warp.calculate_default_transform`, used to calculate the new transformation matrix in the destination CRS, according to the source raster dimensions and bounds. Alternatively, the destination transformation matrix can be obtained from an existing raster; this is common practice when we need to align one raster with another, for instance to be able to combine them in raster algebra operations (Section 3.3.3) (see below)
2. `rasterio.warp.reproject`, introduced in Section 4.3.3, calculates cell values in the destination grid, using the user-selected resampling method (such as nearest neighbor, or bilinear)

Let's take a look at two examples of raster transformation: using categorical and continuous data. Land cover data are usually represented by categorical maps. The `nlcd.tif` file provides information for a small area in Utah, USA, obtained from National Land Cover Database 2011 in the NAD83 / UTM zone 12N CRS. We already created a connection to the `nlcd.tif` file at the beginning of this chapter, named `src_nlcd`.

```
src_nlcd
```

```
<open DatasetReader name='data/nlcd.tif' mode='r'>
```

Recall from previous chapters that the raster transformation matrix and dimensions are accessible from the file connection using `src_nlcd.transform`, `src_nlcd.width`, `src_nlcd.height`, and `src_nlcd.bounds`, respectively.

This information will be required to calculate the destination transformation matrix.

First, let's define the destination CRS. In this case, we choose WGS84 (EPSG code 4326).

```
dst_crs = 'EPSG:4326'
```

Now, we are ready to calculate the destination raster transformation matrix (`dst_transform`), and the destination dimensions (`dst_width`, `dst_height`), using `rasterio.warp.calculate_default_transform`, as follows:

```
dst_transform, dst_width, dst_height = rasterio.warp.calculate_default_transform(
    src_nlcd.crs,
    dst_crs,
    src_nlcd.width,
    src_nlcd.height,
    *src_nlcd.bounds
)
```

Here is the result.

```
dst_transform
```

```
Affine(0.00031506316853514724, 0.0, -113.24138811813536,
       0.0, -0.00031506316853514724, 37.51912722777022)
```

```
dst_width
```

```
1244
```

```
dst_height
```

```
1246
```

> **Note**
>
> The * syntax in Python is known as variable-length '*positional* arguments'. It is used to pass a `list` or `tuple` (or other iterables object) to positional arguments of a function.
> For example, in the last code block, *, in `*src_nlcd.bounds`, is used to unpack `src_nlcd.bounds` (an iterable of length 4) to four separate arguments (`left`, `bottom`, `right`, and `top`), which `rasterio.warp.calculate_default_transform` requires in that order. In other words, the expression from the last example:
>
> ```
> rasterio.warp.calculate_default_transform(
>     src_nlcd.crs,
>     dst_crs,
>     src_nlcd.width,
>     src_nlcd.height,
>     *src_nlcd.bounds
> )
> ```

is a shortcut of:

```
rasterio.warp.calculate_default_transform(
    src_nlcd.crs,
    dst_crs,
    src_nlcd.width,
    src_nlcd.height,
    src_nlcd.bounds[0],
    src_nlcd.bounds[1],
    src_nlcd.bounds[2],
    src_nlcd.bounds[3]
)
```

'*Keyword* arguments' is a related technique; see note in Section 4.3.2.

Recall from Section 4.3.3 that resampling using `rasterio.warp.reproject` can take place directly into a 'destination' raster file connection. Therefore, our next step is to create the metadata file used for writing the reprojected raster to file. For convenience, we are taking the metadata of the source raster (`src_nlcd.meta`), making a copy (`dst_kwargs`), and then updating those specific properties that need to be changed. Note that the reprojection process typically creates 'No Data' pixels, even when there were none in the input raster, since the raster orientation changes and the edges need to be 'filled' to get back a rectangular extent. For example, a reprojected raster may appear as a 'tilted' rectangle, inside a larger straight rectangular extent, whereas the margins around the tilted rectangle are inevitably filled with 'No Data' (e.g., the white stripes surrounding the edges in Figure 6.2 (b) are 'No Data' pixels created as a result of reprojection). We need to specify a 'No Data' value of our choice, if there is no existing definition, or keep the existing source raster 'No Data' setting, such as 255 in this case.

```
dst_kwargs = src_nlcd.meta.copy()
dst_kwargs.update({
    'crs': dst_crs,
    'transform': dst_transform,
    'width': dst_width,
    'height': dst_height
})
dst_kwargs
```

```
{'driver': 'GTiff',
 'dtype': 'uint8',
 'nodata': 255.0,
 'width': 1244,
 'height': 1246,
 'count': 1,
 'crs': 'EPSG:4326',
 'transform': Affine(0.00031506316853514724, 0.0, -113.24138811813536,
        0.0, -0.00031506316853514724, 37.51912722777022)}
```

Now, we are ready to create the reprojected raster. Here, reprojection takes place between two file connections, meaning that the raster value arrays are not being read into memory at once. (It is also possible to reproject into an in-memory `ndarray` object.)

To write the reprojected raster, we first create a destination file connection `dst_nlcd`, pointing at the output file path of our choice (`'output/nlcd_4326.tif'`), using the updated metadata object created earlier (`dst_kwargs`):

```
dst_nlcd = rasterio.open('output/nlcd_4326.tif', 'w', **dst_kwargs)
```

Then, we use the `rasterio.warp.reproject` function to calculate and write the reprojection result into the `dst_nlcd` file connection.

```
rasterio.warp.reproject(
    source=rasterio.band(src_nlcd, 1),
    destination=rasterio.band(dst_nlcd, 1),
    src_transform=src_nlcd.transform,
    src_crs=src_nlcd.crs,
    dst_transform=dst_transform,
    dst_crs=dst_crs,
    resampling=rasterio.enums.Resampling.nearest
)
```

Note–like in the example in Section 4.3.3—that the `source` and `destination` accept a 'band' object, created using `rasterio.band`. In this case, there is just one band. If there were more bands, we would have to repeat the procedure for each band, using `i` instead of `1` inside a loop. Finally, we close the file connection so that the data are actually written.

```
dst_nlcd.close()
```

Many properties of the new object differ from the previous one, including the number of columns and rows (and therefore number of cells), resolution (transformed from meters into degrees), and extent, as summarized below by comparing the `.meta` object of the source and destination rasters.

```
src_nlcd.meta
```

```
{'driver': 'GTiff',
 'dtype': 'uint8',
 'nodata': 255.0,
 'width': 1073,
 'height': 1359,
 'count': 1,
 'crs': CRS.from_epsg(26912),
 'transform': Affine(31.530298224786595, 0.0, 301903.344386758,
        0.0, -31.52465870178793, 4154086.47216415)}
```

```
src_nlcd_4326 = rasterio.open('output/nlcd_4326.tif')
src_nlcd_4326.meta
```

```
{'driver': 'GTiff',
 'dtype': 'uint8',
 'nodata': 255.0,
 'width': 1244,
 'height': 1246,
 'count': 1,
 'crs': CRS.from_epsg(4326),
 'transform': Affine(0.00031506316853514724, 0.0, -113.24138811813536,
        0.0, -0.00031506316853514724, 37.51912722777022)}
```

Examining the unique raster values tells us that the new raster has the same categories, plus the value 255 representing 'No Data':

```
np.unique(src_nlcd.read(1))
```

```
array([1, 2, 3, 4, 5, 6, 7, 8], dtype=uint8)
```

```
np.unique(src_nlcd_4326.read(1))
```

```
array([  1,   2,   3,   4,   5,   6,   7,   8, 255], dtype=uint8)
```

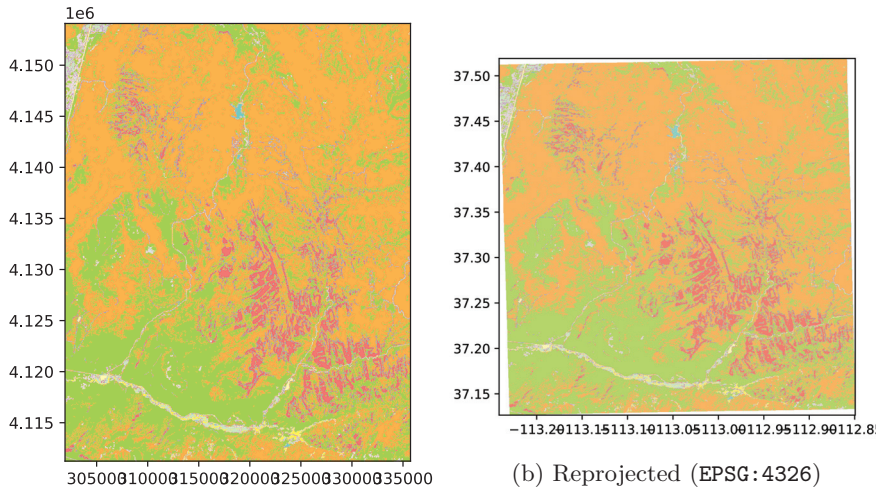
Figure 6.2 illustrates the effect of reprojection, comparing `nlcd.tif` (the input) and `nlcd_4326.tif` (the reprojection result), visually.

```
rasterio.plot.show(src_nlcd, cmap='Set3');
rasterio.plot.show(src_nlcd_4326, cmap='Set3');
```

In the above example, we automatically calculated an optimal (i.e., most information preserving) destination grid using `rasterio.warp.calculate_default_transform`. This is appropriate when there are no specific requirements for the destination raster spatial properties. Namely, we are not required to obtain a specific origin and resolution, but just wish to preserve the raster values as much as possible. To do that, `rasterio.warp.calculate_default_transform` 'tries' to keep the extent and resolution of the destination raster as similar as possible to the source. In other situations, however, we need to reproject a raster into a specific 'template', so that it corresponds, for instance, with other rasters we use in the analysis. In the following code examples, we reproject the `nlcd.tif` raster, again, but this time using the `nlcd_4326.tif` reprojection result as the 'template' to demonstrate this alternative workflow.

First, we create a connection to our 'template' raster to read its metadata.

```
template = rasterio.open('output/nlcd_4326.tif')
template.meta
```



(a) Original (`EPSG:26912`)

(b) Reprojected (`EPSG:4326`)

Figure 6.2: Reprojecting a categorical raster using nearest neighbor resampling

```
{'driver': 'GTiff',
 'dtype': 'uint8',
 'nodata': 255.0,
 'width': 1244,
 'height': 1246,
 'count': 1,
 'crs': CRS.from_epsg(4326),
 'transform': Affine(0.00031506316853514724, 0.0, -113.24138811813536,
        0.0, -0.00031506316853514724, 37.51912722777022)}
```

Then, we create a write-mode connection to our destination raster, using this exact metadata, meaning that the resampling result is going to have identical properties as the 'template'.

```
dst_nlcd_2 = rasterio.open('output/nlcd_4326_2.tif', 'w', **template.meta)
```

Now, we can resample and write the result with `rasterio.warp.reproject`.

```
rasterio.warp.reproject(
    source=rasterio.band(src_nlcd, 1),
    destination=rasterio.band(dst_nlcd_2, 1),
    src_transform=src_nlcd.transform,
    src_crs=src_nlcd.crs,
    dst_transform=dst_nlcd_2.transform,
    dst_crs=dst_nlcd_2.crs,
    resampling=rasterio.enums.Resampling.nearest
)
dst_nlcd_2.close()
```

Naturally, the outputs of the last two examples—`nlcd_4326.tif` and `nlcd_4326_2.tif`—are identical, as we used the same destination grid and the same source data. We can check it with `np.all`.

```
d = rasterio.open('output/nlcd_4326.tif').read(1) == \
    rasterio.open('output/nlcd_4326_2.tif').read(1)
np.all(d)
```

```
np.True_
```

The difference is that in the first example we calculated the template automatically, using `rasterio.warp.calculate_default_transform`, while in the second example we used an existing raster as the 'template'.

Importantly, when the template raster has much more 'coarse' resolution than the source raster, the `rasterio.enums.Resampling.average` (for continuous rasters) or `rasterio.enums.Resampling.mode` (for categorical rasters) resampling methods should be used, instead of `rasterio.enums.Resampling.nearest`. Otherwise, much of the data will be lost, as the 'nearest' method can capture one-pixel value only for each destination raster pixel.

Reprojecting continuous rasters (with numeric or, in this case, integer values) follows an almost identical procedure. This is demonstrated below with `srtm.tif` from the Shuttle Radar Topography Mission (SRTM), which represents height in meters above sea level (elevation) with the WGS84 CRS.

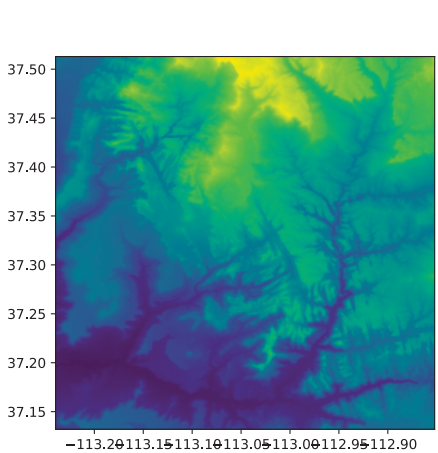
We will reproject this dataset into a projected CRS, but not with the nearest neighbor method. Instead, we will use the bilinear method which computes the output cell value based on the four nearest cells in the original raster. The values in the projected dataset are the distance-weighted average of the values from these four cells: the closer the input cell is to the center of the output cell, the greater its weight. The following code section creates a text string representing WGS 84 / UTM zone 12N, and reprojects the raster into this CRS, using the bilinear method. The code is practically the same as in the first example in this section, except for changing the source and destination file names, and replacing `rasterio.enums.Resampling.nearest` with `rasterio.enums.Resampling.bilinear`.

```
dst_crs = 'EPSG:32612'
dst_transform, dst_width, dst_height = rasterio.warp.calculate_default_transform(
    src_srtm.crs,
    dst_crs,
    src_srtm.width,
    src_srtm.height,
    *src_srtm.bounds
)
dst_kwargs = src_srtm.meta.copy()
dst_kwargs.update({
    'crs': dst_crs,
    'transform': dst_transform,
```

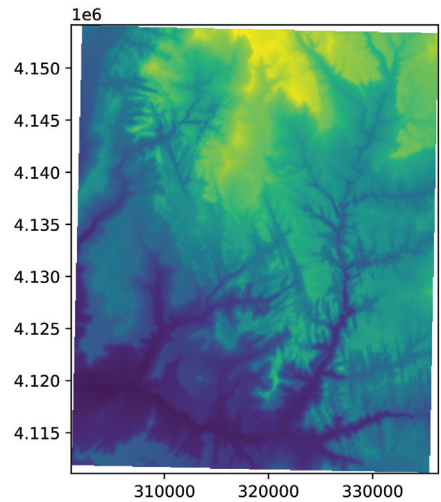
```
    'width': dst_width,
    'height': dst_height
})
dst_srtm = rasterio.open('output/srtm_32612.tif', 'w', **dst_kwargs)
rasterio.warp.reproject(
    source=rasterio.band(src_srtm, 1),
    destination=rasterio.band(dst_srtm, 1),
    src_transform=src_srtm.transform,
    src_crs=src_srtm.crs,
    dst_transform=dst_transform,
    dst_crs=dst_crs,
    resampling=rasterio.enums.Resampling.bilinear
)
dst_srtm.close()
```

Figure 6.3 shows the input and the reprojected SRTM rasters.

```
rasterio.plot.show(src_srtm);
rasterio.plot.show(rasterio.open('output/srtm_32612.tif'));
```



(a) Original (`EPSG:4326`)



(b) Reprojected (`EPSG:32612`)

Figure 6.3: Reprojecting a continuous raster using bilinear resampling

6.9 Custom map projections

Established CRSs captured by `AUTHORITY:CODE` identifiers such as `EPSG:4326` are well suited for many applications. However, it is desirable to use alternative

projections or to create custom CRSs in some cases. Section 6.6 mentioned reasons for using custom CRSs, and provided several possible approaches. Here, we show how to apply these ideas in Python.

One approach is to take an existing WKT definition of a CRS, modify some of its elements, and then use the new definition for reprojecting, using the reprojection methods shown above for vector layers (Section 6.7) and rasters (Section 6.8). For example, let's transform the `zion.gpkg` vector layer to a custom azimuthal equidistant (AEQD) CRS. Using a custom AEQD CRS requires knowing the coordinates of the center point of a dataset in degrees (geographic CRS). In our case, this information can be extracted by calculating the centroid of the `zion` layer transformed into WGS84:

```
lon, lat = zion.to_crs(4326).union_all().centroid.coords[0]
lon, lat
```

```
(-113.02644198455553, 37.298236985233885)
```

Next, we can use the obtained lon/lat coordinates in `coords` to update the WKT definition of the azimuthal equidistant (AEQD) CRS seen below. Notice that we modified just two values below—`"Central_Meridian"` to the longitude and `"Latitude_Of_Origin"` to the latitude of our centroid.

```
my_wkt = f'''PROJCS["Custom_AEQD",
 GEOGCS["GCS_WGS_1984",
  DATUM["WGS_1984",
   SPHEROID["WGS_1984",6378137.0,298.257223563]],
  PRIMEM["Greenwich",0.0],
  UNIT["Degree",0.0174532925199433]],
 PROJECTION["Azimuthal_Equidistant"],
 PARAMETER["Central_Meridian",{lon}],
 PARAMETER["Latitude_Of_Origin",{lat}],
 UNIT["Meter",1.0]]'''
print(my_wkt)
```

```
PROJCS["Custom_AEQD",
 GEOGCS["GCS_WGS_1984",
  DATUM["WGS_1984",
   SPHEROID["WGS_1984",6378137.0,298.257223563]],
  PRIMEM["Greenwich",0.0],
  UNIT["Degree",0.0174532925199433]],
 PROJECTION["Azimuthal_Equidistant"],
 PARAMETER["Central_Meridian",-113.02644198455553],
 PARAMETER["Latitude_Of_Origin",37.298236985233885],
 UNIT["Meter",1.0]]
```

> **Note**
>
> The above expression uses the so-called 'f-strings' syntax, which is one of several Python techniques to embed values inside a string (as alternatives to concatenating with `+`). For example, given:
> `x = 5`
> the expression:
> `f'the value of x is {x}'`
> is a shortcut to:
> `'the value of x is ' + str(x)`
> both returning the string `'the value of x is 5'`.

This approach's last step is to transform our original object (`zion`) to our new custom CRS (`zion_aeqd`).

```
zion_aeqd = zion.to_crs(my_wkt)
```

Custom projections can also be made interactively, for example, using the Projection Wizard[6] web application (Šavrič, Jenny, and Jenny 2016). This website allows you to select a spatial extent of your data and a distortion property, and returns a list of possible projections. The list also contains WKT definitions of the projections that you can copy and use for reprojections. See Open Geospatial Consortium (Open Geospatial Consortium 2019) for details on creating custom CRS definitions with WKT strings.

PROJ strings can also be used to create custom projections, accepting the limitations inherent to projections, especially of geometries covering large geographic areas, as mentioned in Section 6.2. Many projections have been developed and can be set with the `+proj=` element of PROJ strings, with dozens of projections described in detail on the PROJ website alone.

When mapping the world while preserving area relationships, the Mollweide projection, illustrated in Figure 6.4, is a popular and often sensible choice (Jenny et al. 2017). To use this projection, we need to specify it using the proj-string element, `'+proj=moll'`, in the `.to_crs` method:

```
world.to_crs('+proj=moll').plot(color='none', edgecolor='black');
```

It is often desirable to minimize distortion for all spatial properties (area, direction, distance) when mapping the world. One of the most popular projections to achieve this is Winkel tripel (`'+proj=wintri'`), illustrated in Figure 6.5.

```
world.to_crs('+proj=wintri').plot(color='none', edgecolor='black');
```

[6] https://projectionwizard.org/#

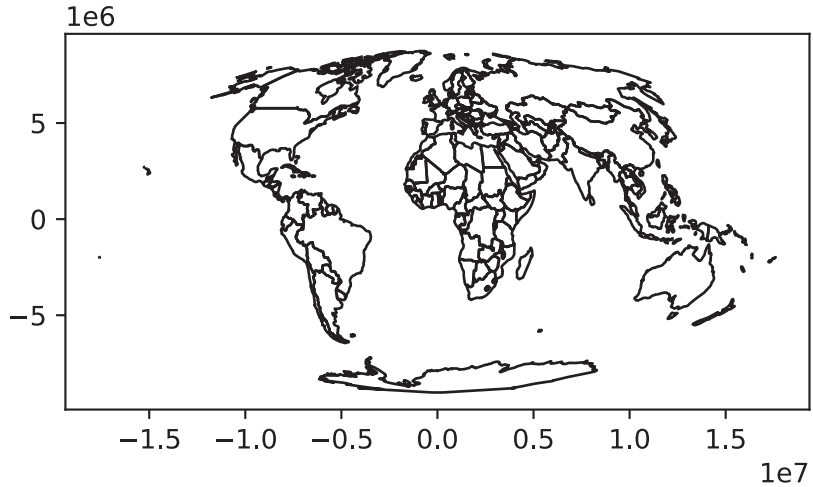


Figure 6.4: Mollweide projection of the world

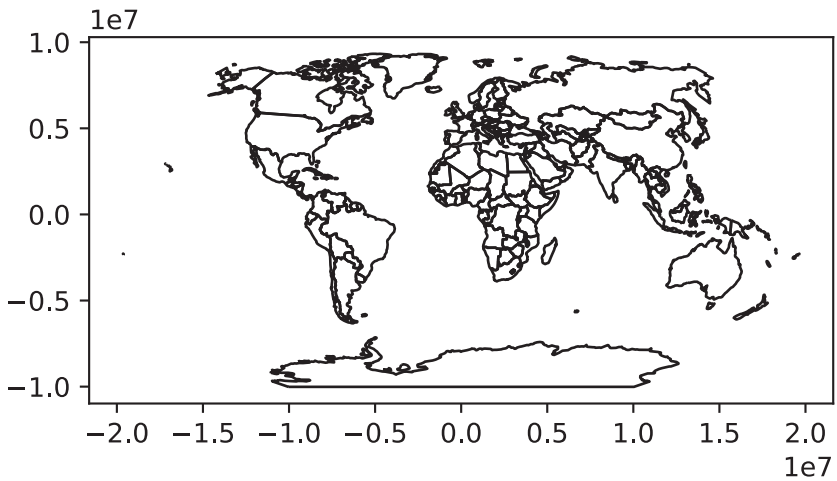


Figure 6.5: Winkel tripel projection of the world

Moreover, proj-string parameters can be modified in most CRS definitions, for example, the center of the projection can be adjusted using the `+lon_0` and `+lat_0` parameters. The below code transforms the coordinates to the Lambert azimuthal equal-area projection centered on the longitude and latitude of New York City (Figure 6.6).

```
world.to_crs('+proj=laea +x_0=0 +y_0=0 +lon_0=-74 +lat_0=40') \
    .plot(color='none', edgecolor='black');
```

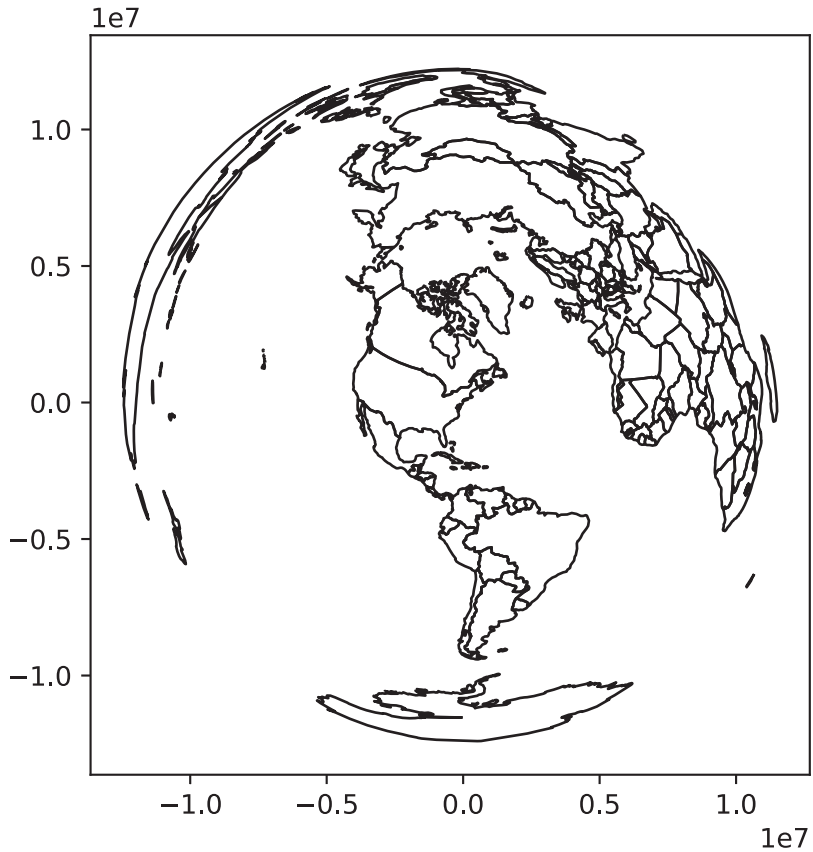


Figure 6.6: Lambert azimuthal equal-area projection of the world centered on New York City

More information on CRS modifications can be found in the Using PROJ documentation[7].

[7] https://proj.org/usage/index.html

7

Geographic data I/O

Prerequisites

This chapter requires importing the following packages:

```
import urllib.request
import zipfile
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import shapely
import pyogrio
import geopandas as gpd
import rasterio
import rasterio.plot
import cartopy
import osmnx as ox
```

It also relies on the following data files:

```
nz = gpd.read_file('data/nz.gpkg')
nz_elev = rasterio.open('data/nz_elev.tif')
```

7.1 Introduction

This chapter is about reading and writing geographic data. Geographic data input is essential for geocomputation: real-world applications are impossible without data. Data output is also vital, enabling others to use valuable new or improved datasets resulting from your work. Taken together, these processes of input/output can be referred to as data I/O.

Geographic data I/O is often done with few lines of code at the beginning and end of projects. It is often overlooked as a simple one-step process. However, mistakes made at the outset of projects (e.g., using an out-of-date or in some way faulty dataset) can lead to large problems later down the line, so it is

DOI: 10.1201/9781003379911-7

worth putting considerable time into identifying which datasets are available, where they can be found and how to retrieve them. These topics are covered in Section 7.2, which describes several geoportals, which collectively contain many terabytes of data, and how to use them. To further ease data access, a number of packages for downloading geographic data have been developed, as demonstrated in Section 7.3.

There are many geographic file formats, each of which has pros and cons, described in Section 7.4. The process of reading and writing files efficiently is covered in Section 7.5 and Section 7.6, respectively.

7.2 Retrieving open data

A vast and ever-increasing amount of geographic data is available on the internet, much of which is free to access and use (with appropriate credit given to its providers)[1]. In some ways there is now too much data, in the sense that there are often multiple places to access the same dataset. Some datasets are of poor quality. In this context, it is vital to know where to look, so the first section covers some of the most important sources. Various 'geoportals' (web services providing geospatial datasets, such as Data.gov[2]) are a good place to start, providing a wide range of data but often only for specific locations (as illustrated in the updated Wikipedia page[3] on the topic).

Some global geoportals overcome this issue. The GEOSS portal[4] and the Copernicus Data Space Ecosystem[5], for example, contain many raster datasets with global coverage. A wealth of vector datasets can be accessed from the SEDAC[6] portal run by the National Aeronautics and Space Administration (NASA) and the European Union's INSPIRE geoportal[7], with global and regional coverage.

Most geoportals provide a graphical interface allowing datasets to be queried based on characteristics such as spatial and temporal extent, the United States Geological Survey's EarthExplorer[8] and NASA's EarthData Search[9] being prime examples. Exploring datasets interactively on a browser is an effective way of understanding available layers. From reproducibility and efficiency perspectives, downloading data is, however, best done with code. Downloads

[1] For example, visit https://freegisdata.rtwilson.com/ for a vast list of websites with freely available geographic datasets.
[2] https://catalog.data.gov/dataset?metadata_type=geospatial
[3] https://en.wikipedia.org/wiki/Geoportal
[4] http://www.geoportal.org/
[5] https://dataspace.copernicus.eu//
[6] http://sedac.ciesin.columbia.edu/
[7] http://inspire-geoportal.ec.europa.eu/
[8] https://earthexplorer.usgs.gov/
[9] https://search.earthdata.nasa.gov/search

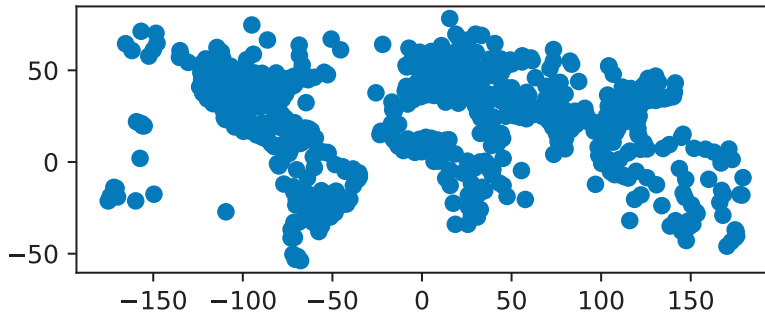


Figure 7.1: World airports layer, downloaded from the Natural Earth Data website using Python

can be initiated from the command line using a variety of techniques, primarily via URLs and APIs (see the Sentinel API[10], for example).

Files hosted on static URLs can be downloaded with the following method, as illustrated in the code chunk below which accesses the Natural Earth Data[11] website to download the world airports layer zip file and to extract the contained ESRI Shapefile. Note that the download code is complicated by the fact that the server checks the `User-agent` header of the request, basically to make sure that the download takes place through a browser. To overcome this, we add a header corresponding to a request coming from a browser (such as Firefox) in our code.

```
# Set URL+filename
url = 'https://naciscdn.org/naturalearth/10m/cultural/ne_10m_airports.zip'
filename = 'output/ne_10m_airports.zip'
# Download
urllib.request.urlretrieve(url, filename)
# Extract
f = zipfile.ZipFile(filename, 'r')
f.extractall('output')
f.close()
```

The ESRI Shapefile that has been created in the `output` directory can then be imported and plotted (Figure 7.1) as follows using **geopandas**.

[10] https://scihub.copernicus.eu/twiki/do/view/SciHubWebPortal/APIHubDescription
[11] https://www.naturalearthdata.com/

```
ne = gpd.read_file(filename.replace('.zip', '.shp'))
ne.plot();
```

7.3 Geographic data packages

Several Python packages have been developed for accessing geographic data, two of which are demonstrated below. These provide interfaces to one or more spatial libraries or geoportals and aim to make data access even quicker from the command line.

Administrative borders are often useful in spatial analysis. These can be accessed with the `cartopy.io.shapereader.natural_earth` function from the **cartopy** package (Met Office 2010-2015). For example, the following code loads the `'admin_2_counties'` dataset of US counties into a `GeoDataFrame`.

```
filename = cartopy.io.shapereader.natural_earth(
    resolution='10m',
    category='cultural',
    name='admin_2_counties'
)
counties = gpd.read_file(filename)
counties
```

	FEATURECLA	SCALERANK	...	NAME_ZHT	geometry
0	Admin-2 scale rank	0	...	霍特科姆縣	MULTIPOLYGON (((-122.75302 48.9...
1	Admin-2 scale rank	0	...	奧卡諾根縣	POLYGON ((-120.85196 48.99251, ...
2	Admin-2 scale rank	0	...	費里縣	POLYGON ((-118.83688 48.99251, ...
...	...	...	...	...	...
3221	Admin-2 scale rank	0	...	維拉爾巴	POLYGON ((-66.44407 18.17665, -...
3222	Admin-2 scale rank	0	...	大薩瓦納	POLYGON ((-66.88464 18.02481, -...
3223	Admin-2 scale rank	0	...	馬里考	POLYGON ((-66.89856 18.1879, -6...

The resulting layer `counties` is shown in Figure 7.2.

```
counties.plot();
```

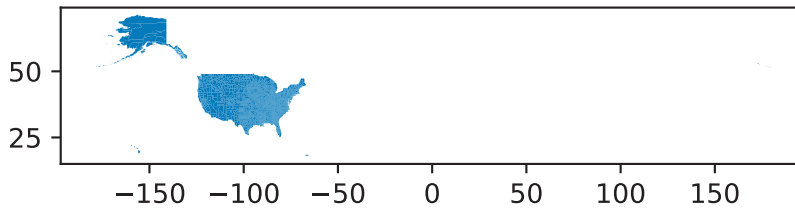


Figure 7.2: US counties, downloaded from the Natural Earth Data website using package **cartopy**

Note that Figure 7.2 x-axis spans the entire range of longitudes, between `-180` and `180`, since the Aleutian Islands county (which is small and difficult to see on the map) crosses the International Date Line.

Other layers can from NaturalEarth be accessed the same way. You need to specify the `resolution`, `category`, and `name` of the requested dataset in Natural Earth Data, then run the `cartopy.io.shapereader.natural_earth`, which downloads the file(s) and returns the path, and read the file into the Python environment, e.g., using `gpd.read_file`. This is an alternative approach to 'directly' downloading files as shown earlier (Section 7.2).

The second example uses the **osmnx** package (Boeing 2017) to find parks from the OpenStreetMap (OSM) database. As illustrated in the code chunk below, OpenStreetMap data can be obtained using the `ox.features.features_from_place` function. The first argument is a string which is geocoded to a polygon (the `ox.features.features_from_bbox` and `ox.features.features_from_polygon` can also be used to query a custom area of interest). The second argument specifies the OSM tag(s)[12], selecting which OSM elements we're interested in (parks, in this case), represented by key-value pairs.

```
parks = ox.features.features_from_place(
    query='leeds uk',
    tags={'leisure': 'park'}
)
```

The result is a `GeoDataFrame` with the parks in Leeds. Now, we can plot the geometries with the `name` property in the tooltips using `explore` (Figure 7.3).

```
parks[['name', 'geometry']].explore()
```

[12] https://wiki.openstreetmap.org/wiki/Map_features

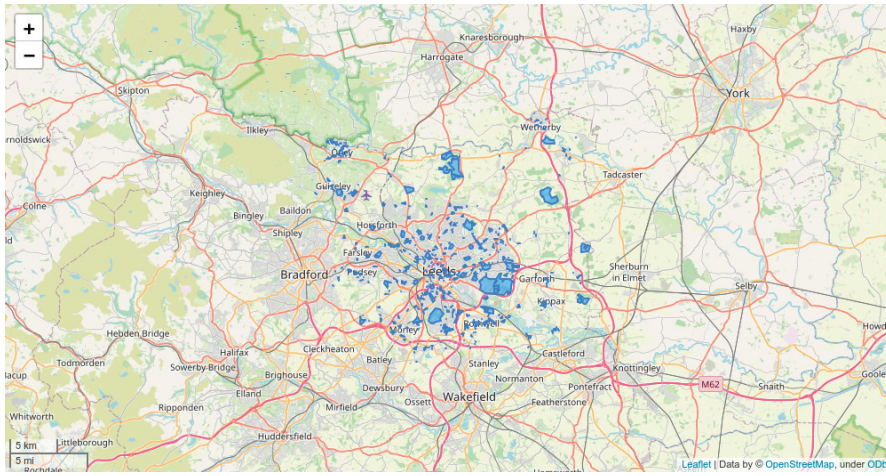


Figure 7.3: Parks in Leeds, based on OpenStreetMap data, downloaded using package **osmnx**

It should be noted that the **osmnx** package downloads OSM data from the Overpass API[13], which is rate limited and therefore unsuitable for queries covering very large areas. To overcome this limitation, you can download OSM data extracts, such as in Shapefile format from Geofabrik[14], and then load them from the file into the Python environment.

OpenStreetMap is a vast global database of crowd-sourced data, is growing daily, and has a wider ecosystem of tools enabling easy access to the data, from the Overpass turbo[15] web service for rapid development and testing of OSM queries to `osm2pgsql` for importing the data into a PostGIS database. Although the quality of datasets derived from OSM varies, the data source and wider OSM ecosystems have many advantages: they provide datasets that are available globally, free of charge, and constantly improving thanks to an army of volunteers. Using OSM encourages 'citizen science' and contributions back to the digital commons (you can start editing data representing a part of the world you know well at https://www.openstreetmap.org/).

One way to obtain spatial information is to perform geocoding—transform a description of a location, usually an address, into a set of coordinates. This is typically done by sending a query to an online service and getting the location as a result. Many such services exist that differ in the used method of

[13] https://wiki.openstreetmap.org/wiki/Overpass_API
[14] https://download.geofabrik.de/
[15] https://overpass-turbo.eu/

geocoding, usage limitations, costs, or API key requirements. Nominatim[16] is a well-known free service, based on OpenStreetMap data, and there are many other free and commercial geocoding services.

geopandas provides the `gpd.tools.geocode` function, which can geocode addresses to a `GeoDataFrame`. Internally it uses the **geopy** package, supporting several providers through the `provider` parameter (use `geopy.geocoders.SERVICE_TO_GEOCODER` to see possible options). The example below searches for John Snow blue plaque[17] coordinates located on a building in the Soho district of London. The result is a `GeoDataFrame` with the address we passed to `gpd.tools.geocode`, and the detected point location.

```
result = gpd.tools.geocode('54 Frith St, London W1D 4SJ, UK', timeout=10)
result
```

	geometry	address
0	POINT (-0.13178 51.51377)	54, Frith Street, W1D 3JD, Frit...

Importantly, (1) we can pass a `list` of multiple addresses instead of just one, resulting in a `GeoDataFrame` with corresponding multiple rows, and (2) 'No Results' responses are represented by `POINT EMPTY` geometries, as shown in the following example.

```
result = gpd.tools.geocode(
    ['54 Frith St, London W1D 4SJ, UK', 'abcdefghijklmnopqrstuvwxyz'],
    timeout=10
)
result
```

	geometry	address
0	POINT (-0.13178 51.51377)	54, Frith Street, W1D 3JD, Frit...
1	POINT EMPTY	None

The result is visualized in Figure 7.4 using the `.explore` function. We are using the `marker_kwds` parameter of `.explore` to make the marker larger (see Section 8.3.2).

```
result.iloc[[0]].explore(color='red', marker_kwds={'radius':20})
```

[16] https://nominatim.openstreetmap.org/ui/about.html
[17] https://en.m.wikipedia.org/wiki/John_Snow_(public_house)

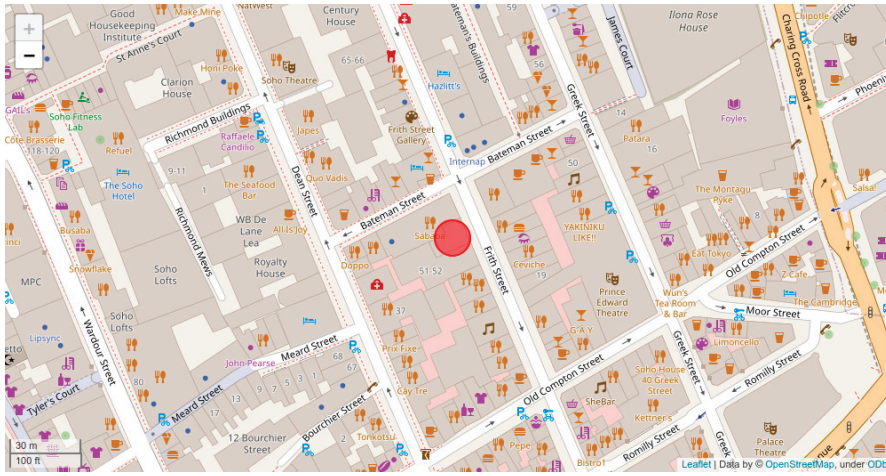


Figure 7.4: Specific address in London, geocoded into a `GeoDataFrame`

7.4 File formats

Geographic datasets are usually stored as files or in spatial databases. File formats usually can either store vector or raster data, while spatial databases such as PostGIS can store both. The large variety of file formats may seem bewildering, but there has been much consolidation and standardization since the beginnings of GIS software in the 1960s when the first widely distributed program SYMAP for spatial analysis was created at Harvard University (Coppock and Rhind 1991).

GDAL (which originally was pronounced as 'goo-dal', with the double 'o' making a reference to object-orientation), the Geospatial Data Abstraction Library, has resolved many issues associated with incompatibility between geographic file formats since its release in 2000. GDAL provides a unified and high-performance interface for reading and writing of many raster and vector data formats. Many open and proprietary GIS programs, including GRASS, ArcGIS and QGIS, use GDAL behind their GUIs for doing the legwork of ingesting and spitting out geographic data in appropriate formats. Most Python packages for working with spatial data, including **geopandas** and **rasterio** used in this book, also rely on GDAL for importing and exporting spatial data files.

GDAL provides access to more than 200 vector and raster data formats. Table 7.1 presents some basic information about selected and often-used spatial file formats.

Table 7.1: Commonly used spatial data file formats

Name	Extension	Info	Type	Model
ESRI Shapefile	`.shp` (the main file)	Popular format consisting of at least three files. No support for: files > 2GB; mixed types; names > 10 chars; cols > 255.	Vector	Partially open
GeoJSON	`.geojson`	Extends the JSON exchange format by including a subset of the simple feature representation; mostly used for storing coordinates in longitude and latitude; it is extended by the TopoJSON format.	Vector	Open
KML	`.kml`	XML-based format for spatial visualization, developed for use with Google Earth. Zipped KML file forms the KMZ format.	Vector	Open
GPX	`.gpx`	XML schema created for exchange of GPS data.	Vector	Open
FlatGeobuf	`.fgb`	Single file format allowing for quick reading and writing of vector data. Has streaming capabilities.	Vector	Open
GeoTIFF	`.tif/.tiff`	Popular raster format. A TIFF file containing additional spatial metadata.	Raster	Open
Arc ASCII	`.asc`	Text format where the first six lines represent the raster header, followed by the raster cell values arranged in rows and columns.	Raster	Open
SQLite/SpatiaLite	`.sqlite`	Standalone relational database, SpatiaLite is the spatial extension of SQLite.	Vector and raster	Open
ESRI FileGDB	`.gdb`	Spatial and nonspatial objects created by ArcGIS. Allows: multiple feature classes; topology. Limited support from GDAL.	Vector and raster	Proprietary
GeoPackage	`.gpkg`	Lightweight database container based on SQLite allowing an easy and platform-independent exchange of geodata.	Vector and (very limited) raster	Open

An important development ensuring the standardization and open-sourcing of file formats was the founding of the Open Geospatial Consortium (OGC) in 1994. Beyond defining the Simple Features data model (see Section 1.2.4), the OGC also coordinates the development of open standards, for example as used in file formats such as KML and GeoPackage. Open file formats of the kind endorsed by the OGC have several advantages over proprietary formats:

the standards are published, ensure transparency and open up the possibility for users to further develop and adjust the file formats to their specific needs.

ESRI Shapefile is the most popular vector data exchange format; however, it is not a fully open format (though its specification is open). It was developed in the early 1990s and, from a modern standpoint, has a number of limitations. First of all, it is a multi-file format, which consists of at least three files. It also only supports 255 columns, its column names are restricted to ten characters and the file size limit is 2 GB. Furthermore, ESRI Shapefile does not support all possible geometry types, for example, it is unable to distinguish between a polygon and a multipolygon. Despite these limitations, a viable alternative had been missing for a long time. In 2014, GeoPackage emerged, and seems to be a more than suitable replacement candidate for ESRI Shapefile. GeoPackage is a format for exchanging geospatial information and an OGC standard. This standard describes the rules on how to store geospatial information in a tiny SQLite container. Hence, GeoPackage is a lightweight spatial database container, which allows the storage of vector and raster data but also of non-spatial data and extensions. Aside from GeoPackage, there are other geospatial data exchange formats worth checking out (Table 7.1).

The GeoTIFF format seems to be the most prominent raster data format. It allows spatial information, such as the CRS definition and the transformation matrix (see Section 1.3.1), to be embedded within a TIFF file. Similar to ESRI Shapefile, this format was firstly developed in the 1990s, but as an open format. Additionally, GeoTIFF is still being expanded and improved. One of the most significant recent additions to the GeoTIFF format is its variant called COG (Cloud Optimized GeoTIFF). Raster objects saved as COGs can be hosted on HTTP servers, so other people can read only parts of the file without downloading the whole file (Section 7.5.2).

There is also a plethora of other spatial data formats that we do not explain in detail or mention in Table 7.1 due to the book limits. If you need to use other formats, we encourage you to read the GDAL documentation about vector and raster drivers. Additionally, some spatial data formats can store other data models (types) than vector or raster. Two examples are LAS and LAZ formats for storing lidar point clouds, and NetCDF and HDF for storing multidimensional arrays.

Finally, spatial data are also often stored using tabular (non-spatial) text formats, including CSV files or Excel spreadsheets. This can be convenient to share spatial (point) datasets with people who, or software that, struggle with spatial data formats. If necessary, the table can be converted to a point layer (see examples in Section 1.2.6 and Section 3.2.3).

7.5 Data input (I)

Executing commands such as `gpd.read_file` (the main function we use for loading vector data) or `rasterio.open+.read` (the main group of functions used for loading raster data) silently sets off a chain of events that reads data from files. Moreover, there are many Python packages containing a wide range of geographic data or providing simple access to different data sources. All of them load the data into the Python environment or, more precisely, assign objects to your workspace, stored in RAM and accessible within the Python session. The latter is the most straightforward approach, suitable when RAM is not a limiting factor. For large vector layers and rasters, partial reading may be required. For vector layers, we will demonstrate how to read subsets of vector layers, filtered by attributes or by location (Section 7.5.1). For rasters, we already showed earlier in the book how the user can choose which specific bands to read (Section 1.3.1), or read resampled data to a lower resolution (Section 4.3.2). In this section, we also show how to read specific rectangular extents ('windows') from a raster file (Section 7.5.2).

7.5.1 Vector data

Spatial vector data comes in a wide variety of file formats. Most popular representations such as `.shp`, `.geojson`, and `.gpkg` files can be imported and exported with **geopandas** function `gpd.read_file` and method `.to_file` (covered in Section 7.6), respectively.

geopandas uses GDAL to read and write data, via **pyogrio** since `geopandas` version `1.0.0` (previously via **fiona**). After **pyogrio** is imported, `pyogrio.list_drivers` can be used to list drivers available to GDAL, including whether they can read (`'r'`), append (`'a'`), or write (`'w'`) data, or all three.

```
pyogrio.list_drivers()
```

```
{'PCIDSK': 'rw',
 'PDS4': 'rw',
 ...
 'AVCE00': 'r',
 'HTTP': 'r'}
```

The first argument of the **geopandas** versatile data import function `gpd.read_file` is `filename`, which is typically a string, but can also be a file connection. The content of a string could vary between different drivers. In most cases, as with the ESRI Shapefile (`.shp`) or the GeoPackage format (`.gpkg`), the `filename` argument would be a path or a URL to an actual file,

such as `geodata.gpkg`. The driver is automatically selected based on the file extension, as demonstrated for a `.gpkg` file below.

```
world = gpd.read_file('data/world.gpkg')
```

For some drivers, such as a File Geodatabase (`OpenFileGDB`), `filename` could be provided as a folder name. GeoJSON, a plain text format, on the other hand, can be read from a `.geojson` file, but also from a string.

```
gpd.read_file('{"type":"Point","coordinates":[34.838848,31.296301]}')
```

	geometry
0	POINT (34.83885 31.2963)

Some vector formats, such as GeoPackage, can store multiple data layers. By default, `gpd.read_file` reads the first layer of the file specified in `filename`. However, using the `layer` argument you can specify any other layer. To list the available layers, we can use function `gpd.list_layers` (or `pyogrio.list_layers`).

The `gpd.read_file` function also allows for reading just parts of the file into RAM with two possible mechanisms. The first one is related to the `where` argument, which allows specifying what part of the data to read using an SQL `WHERE` expression. An example below extracts data for Tanzania only from the `world.gpkg` file (Figure 7.5 (a)). It is done by specifying that we want to get all rows for which `name_long` equals to `'Tanzania'`.

```
tanzania = gpd.read_file('data/world.gpkg', where='name_long="Tanzania"')
tanzania
```

	iso_a2	name_long	...	gdpPercap	geometry
0	TZ	Tanzania	...	2402.099404	MULTIPOLYGON (((33.90371 -0.95,...

If you do not know the names of the available columns, a good approach is to read the layer metadata using `pyogrio.read_info`. The resulting object contains, among other properties, the column names (`fields`) and data types (`dtypes`):

```
info = pyogrio.read_info('data/world.gpkg')
info['fields']
```

```
array(['iso_a2', 'name_long', 'continent', 'region_un', 'subregion',
       'type', 'area_km2', 'pop', 'lifeExp', 'gdpPercap'], dtype=object)
```

```
info['dtypes']
```

```
array(['object', 'object', 'object', 'object', 'object', 'object',
       'float64', 'float64', 'float64', 'float64'], dtype=object)
```

The second mechanism uses the `mask` argument to filter data based on intersection with an existing geometry. This argument expects a geometry (`GeoDataFrame`, `GeoSeries`, or `shapely` geometry) representing the area where we want to extract the data. Let's try it using a small example—we want to read polygons from our file that intersect with the buffer of 50,000 m of Tanzania's borders. To do it, we need to transform the geometry to a projected CRS (such as `EPSG:32736`), prepare our 'filter' by creating the buffer (Section 4.2.3), and transform back to the original CRS to be used as a mask (Figure 7.5 (a)).

```
tanzania_buf = tanzania.to_crs(32736).buffer(50000).to_crs(4326)
```

Now, we can pass the 'filter' geometry `tanzania_buf` to the `mask` argument of `gpd.read_file`.

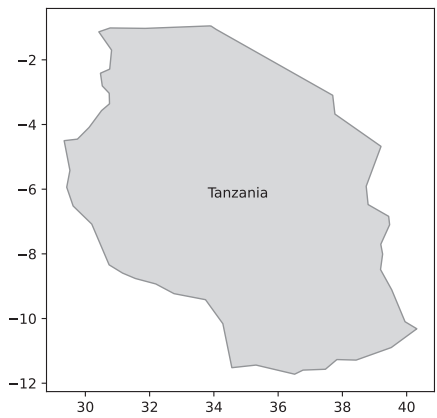
```
tanzania_neigh = gpd.read_file('data/world.gpkg', mask=tanzania_buf)
```

Our result, shown in Figure 7.5 (b), contains Tanzania and every country intersecting with its 50,000 m buffer. Note that the last two expressions are used to add text labels with the `name_long` of each country, placed at the country centroid.

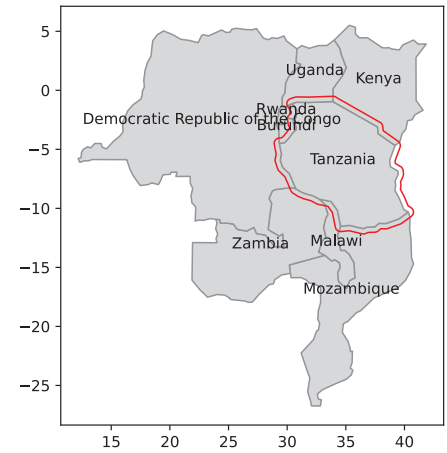
```
# Using 'where'
fig, ax = plt.subplots()
tanzania.plot(ax=ax, color='lightgrey', edgecolor='grey')
tanzania.apply(
    lambda x: ax.annotate(text=x['name_long'],
    xy=x.geometry.centroid.coords[0], ha='center'), axis=1
);
# Using 'mask'
fig, ax = plt.subplots()
tanzania_neigh.plot(ax=ax, color='lightgrey', edgecolor='grey')
tanzania_buf.plot(ax=ax, color='none', edgecolor='red')
tanzania_neigh.apply(
    lambda x: ax.annotate(text=x['name_long'],
    xy=x.geometry.centroid.coords[0], ha='center'), axis=1
);
```

A different, `gpd.read_postgis`, function can be used to read a vector layer from a PostGIS database.

Often we need to read CSV files (or other tabular formats) which have x and y coordinate columns, and turn them into a `GeoDataFrame` with point geometries. To do that, we can import the file using **pandas** (e.g., using `pd.read_csv` or `pd.read_excel`), then go from `DataFrame` to `GeoDataFrame` using the `gpd.points_from_xy` function, as shown earlier in the book (See Section 1.2.6 and Section 3.2.3). For example, the table `cycle_hire_xy.csv`, where the coordinates are stored in the `X` and `Y` columns in `EPSG:4326`, can be imported, converted to a `GeoDataFrame`, and plotted, as follows (Figure 7.6).



(a) Using a `where` query (matching `'Tanzania'`)



(b) Using a `mask` (a geometry shown in red)

Figure 7.5: Reading a subset of the vector layer file `world.gpkg`

```
cycle_hire = pd.read_csv('data/cycle_hire_xy.csv')
geom = gpd.points_from_xy(cycle_hire['X'], cycle_hire['Y'], crs=4326)
geom = gpd.GeoSeries(geom)
cycle_hire_xy = gpd.GeoDataFrame(data=cycle_hire, geometry=geom)
cycle_hire_xy.plot();
```

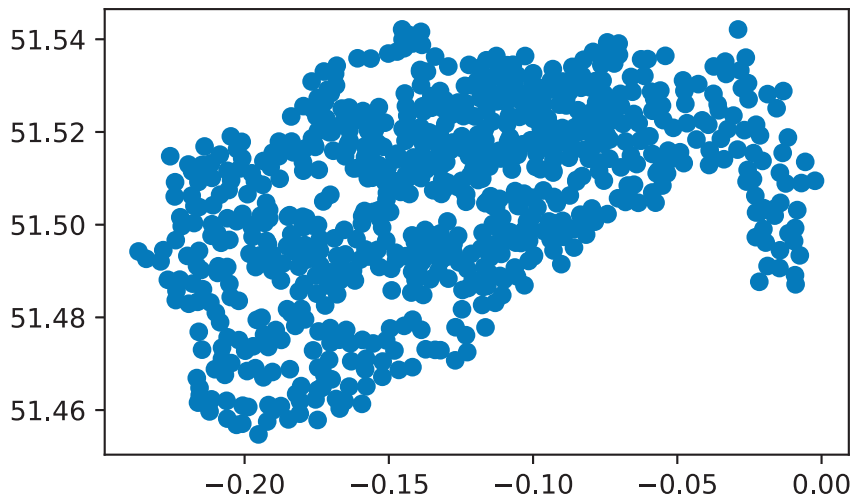


Figure 7.6: The `cycle_hire_xy.csv` table transformed to a point layer

Instead of columns describing 'XY' coordinates, a single column can also contain the geometry information, not necessarily points but possibly any other geometry type. Well-known text (WKT), well-known binary (WKB), and GeoJSON are examples of formats used to encode geometry in such a column. For instance, the `world_wkt.csv` file has a column named `'WKT'`, representing polygons of the world's countries (in WKT format). When importing the CSV file into a `DataFrame`, the `'WKT'` column is interpreted just like any other string column.

```
world_wkt = pd.read_csv('data/world_wkt.csv')
world_wkt
```

	WKT	iso_a2	...	lifeExp	gdpPercap
0	MULTIPOLYGON (((180.0 -16.06713...	FJ	...	69.960000	8222.253784
1	MULTIPOLYGON (((33.903711197104...	TZ	...	64.163000	2402.099404
2	MULTIPOLYGON (((-8.665589565454...	EH	...	NaN	NaN
...	...	...	...	...	...
174	MULTIPOLYGON (((20.590246546680...	XK	...	71.097561	8698.291559
175	MULTIPOLYGON (((-61.68 10.76,-6...	TT	...	70.426000	31181.821196
176	MULTIPOLYGON (((30.833852421715...	SS	...	55.817000	1935.879400

To convert it to a `GeoDataFrame`, we can apply the `gpd.GeoSeries.from_wkt` function (which is analogous to `shapely`'s `shapely.from_wkt`, see Section 1.2.5) on the WKT strings, to convert the series of WKT strings into a `GeoSeries` with the geometries.

```
world_wkt['geometry'] = gpd.GeoSeries.from_wkt(world_wkt['WKT'])
world_wkt = gpd.GeoDataFrame(world_wkt)
world_wkt
```

	WKT	iso_a2	...	gdpPercap	geometry
0	MULTIPOLYGON (((180.0 -16.06713...	FJ	...	8222.253784	MULTIPOLYGON (((180 -16.06713, ...
1	MULTIPOLYGON (((33.903711197104...	TZ	...	2402.099404	MULTIPOLYGON (((33.90371 -0.95,...
2	MULTIPOLYGON (((-8.665589565454...	EH	...	NaN	MULTIPOLYGON (((-8.66559 27.656...
...	...	...	...	...	...
174	MULTIPOLYGON (((20.590246546680...	XK	...	8698.291559	MULTIPOLYGON (((20.59025 41.855...
175	MULTIPOLYGON (((-61.68 10.76,-6...	TT	...	31181.821196	MULTIPOLYGON (((-61.68 10.76, -...
176	MULTIPOLYGON (((30.833852421715...	SS	...	1935.879400	MULTIPOLYGON (((30.83385 3.5091...

The resulting layer is shown in Figure 7.7.

```
world_wkt.plot();
```

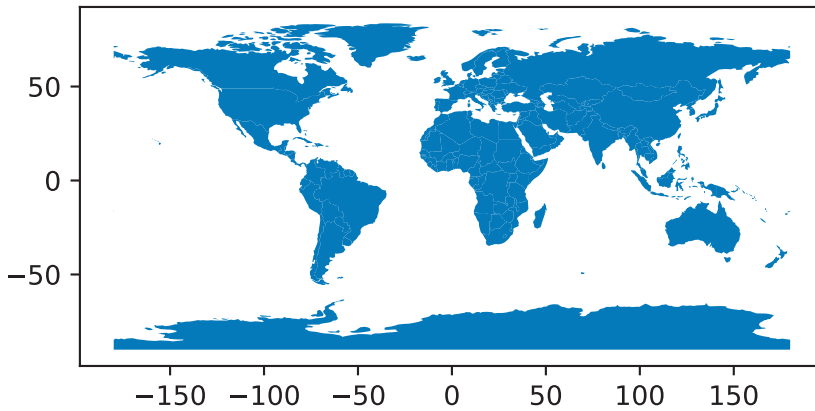


Figure 7.7: The `world_wkt.csv` table transformed to a polygon layer

As a final example, we will show how **geopandas** also reads KML files. A KML file stores geographic information in XML format—a data format for the creation of web pages and the transfer of data in an application-independent way (Nolan and Lang 2014). Here, we access a KML file from the web.

The sample KML file `KML_Samples.kml` contains more than one layer.

```
u = 'https://developers.google.com/kml/documentation/KML_Samples.kml'
gpd.list_layers(u)
```

	name	geometry_type
0	Placemarks	Point Z
1	Highlighted Icon	Point Z
2	Paths	LineString Z
3	Google Campus	Polygon Z
4	Extruded Polygon	Polygon Z
5	Absolute and Relative	Polygon Z

We can choose, for instance, the first layer `'Placemarks'` and read it, using `gpd.read_file` with an additional `layer` argument.

```
placemarks = gpd.read_file(u, layer='Placemarks')
placemarks
```

	Name	Description	geometry
0	Simple placemark	Attached to the ground. Intelli...	POINT Z (-122.0822 37.42229 0)
1	Floating placemark	Floats a defined distance above...	POINT Z (-122.08408 37.422 50)
2	Extruded placemark	Tethered to the ground by a cus...	POINT Z (-122.08577 37.42157 50)

7.5.2 Raster data

Similar to vector data, raster data comes in many file formats, some of which support multilayer files. `rasterio.open` is used to create a file connection to a raster file, which can be subsequently used to read the metadata and/or the values, as shown previously (Section 1.3.1).

```
src = rasterio.open('data/srtm.tif')
src
```

```
<open DatasetReader name='data/srtm.tif' mode='r'>
```

All of the previous examples, like the one above, read spatial information from files stored on your hard drive. However, GDAL also allows reading data directly from online resources, such as HTTP/HTTPS/FTP web resources. Let's try it by connecting to the global monthly snow probability at 500 m resolution for the period 2000-2012 (Hengl 2021). Snow probability for December is stored as a Cloud Optimized GeoTIFF (COG) file (see Section 7.4) and can be accessed by its HTTPS URI.

```
url = 'https://zenodo.org/record/5774954/files/'
url += 'clm_snow.prob_esacci.dec_p.90_500m_s0..0cm_2000..2012_v2.0.tif'
src = rasterio.open(url)
src
```

```
<open DatasetReader name='https://zenodo.org/record/5774954/
files/clm_snow.prob_esacci.dec_p.90_500m_s0..0cm_2000..2012_
v2.0.tif' mode='r'>
```

In the example above `rasterio.open` creates a connection to the file without obtaining any values, as we did for the local `srtm.tif` file. The values can be read into an `ndarray` using the `.read` method of the file connection (Section 1.3.1). Using parameters of `.read` allows us to just read a small portion of the data, without downloading the entire file. This is very useful when working with large datasets hosted online from resource-constrained computing environments such as laptops.

For example, we can read a specified rectangular extent of the raster. With **rasterio**, this is done using the so-called *windowed reading* capabilities. Note that, with windowed reading, we import just a subset of the raster extent into an `ndarray` covering any partial extent. Windowed reading is therefore

memory- (and, in this case, bandwidth-) efficient, since it avoids reading the entire raster into memory. It can also be considered an alternative pathway to *cropping* (Section 5.2).

To read a raster *window*, let's first define the bounding box coordinates. For example, here we use a 10×10 degrees extent coinciding with Reykjavik.

```
xmin=-30
xmax=-20
ymin=60
ymax=70
```

Using the extent coordinates along with the raster transformation matrix, we create a window object, using the `rasterio.windows.from_bounds` function. This function basically 'translates' the extent from coordinates, to row/column ranges.

```
w = rasterio.windows.from_bounds(
    left=xmin,
    bottom=ymin,
    right=xmax,
    top=ymax,
    transform=src.transform
)
w
```

```
Window(col_off=35999.99999999998, row_off=4168.799999999996,
width=2399.9999999999927, height=2400.0)
```

Now we can read the partial array, according to the specified window `w`, by passing it to the `.read` method.

```
r = src.read(1, window=w)
r
```

```
array([[100, 100, 100, ..., 255, 255, 255],
       [100, 100, 100, ..., 255, 255, 255],
       [100, 100, 100, ..., 255, 255, 255],
       ...,
       [255, 255, 255, ..., 255, 255, 255],
       [255, 255, 255, ..., 255, 255, 255],
       [255, 255, 255, ..., 255, 255, 255]], dtype=uint8)
```

Note that the transformation matrix of the window is not the same as that of the original raster (unless it incidentally starts from the top-left corner)! Therefore, we must re-create the transformation matrix, with the modified origin (`xmin`,`ymax`), yet the same resolution, as follows.

```
w_transform = rasterio.transform.from_origin(
    west=xmin,
    north=ymax,
    xsize=src.transform[0],
    ysize=abs(src.transform[4])
)
w_transform
```

```
Affine(0.00416666666666667, 0.0, -30.0,
       0.0, -0.00416666666666667, 70.0)
```

The array `r` along with the updated transformation matrix `w_transform` comprise the partial window, which we can keep working with just like with any other raster, as shown in previous chapters. Figure 7.8 shows the result, along with the location of Reykjavik.

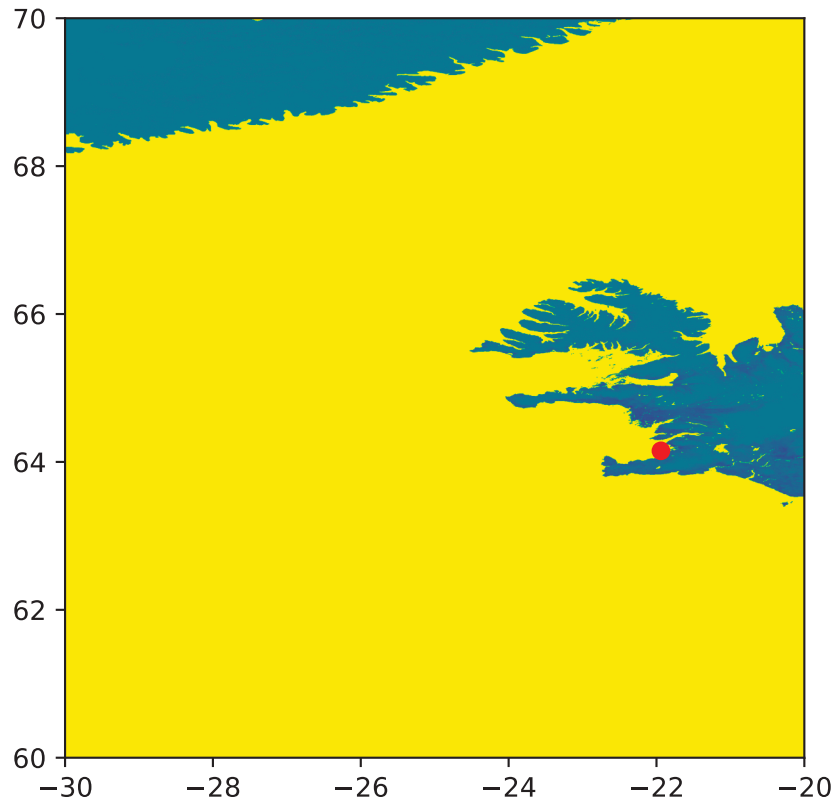


Figure 7.8: Raster window read from a remote Cloud Optimized GeoTIFF (COG) file source

```
fig, ax = plt.subplots()
rasterio.plot.show(r, transform=w_transform, ax=ax)
gpd.GeoSeries(shapely.Point(-21.94, 64.15)).plot(ax=ax, color='red');
```

Another option is to extract raster values at particular points, directly from the file connection, using the `.sample` method (see Section 3.3.1). For example, we can get the snow probability for December in Reykjavik (70%) by specifying its coordinates and applying `.sample`.

```
coords = (-21.94, 64.15)
values = src.sample([coords])
list(values)
```

```
[array([70], dtype=uint8)]
```

The example above efficiently extracts and downloads a single value instead of the entire GeoTIFF file, saving valuable resources.

Note that URIs can also identify *vector* datasets, enabling you to import datasets from online storage with **geopandas**, including datasets within ZIP archives hosted on the web.

```
gpd.read_file("zip+https://github.com/Toblerity/Fiona/files/11151652/coutwildrnp.zip")
```

	PERIMETER	FEATURE2	...	STATE	geometry
0	1.221070	None	...	UT	POLYGON ((-111.73528 41.99509, ...
1	0.755827	None	...	UT	POLYGON ((-112.00385 41.5527, -...
2	1.708510	None	...	CO	POLYGON ((-106.79289 40.98353, ...
...	...	...	...	...	...
64	0.263251	None	...	CO	POLYGON ((-108.35329 37.26869, ...
65	0.119581	None	...	CO	POLYGON ((-108.44212 37.29754, ...
66	0.120627	None	...	CO	POLYGON ((-108.5527 37.28285, -...

7.6 Data output (O)

Writing geographic data allows you to convert from one format to another and to save newly created objects for permanent storage. Depending on the data type (vector or raster), object class (e.g., `GeoDataFrame`), and type and amount of stored information (e.g., object size, range of values), it is important to know how to store spatial files in the most efficient way. The next two subsections will demonstrate how to do this.

7.6.1 Vector data

The counterpart of `gpd.read_file` is the `.to_file` method that a `GeoDataFrame` has. It allows you to write `GeoDataFrame` objects to a wide range of geographic vector file formats, including the most common ones, such as `.geojson`, `.shp` and `.gpkg`. Based on the file name, `.to_file` decides automatically which driver to use. The speed of the writing process depends also on the driver.

For example, to export the `world` layer to a GeoPackage file, we can use `.to_file` and specify the output file name.

```
world.to_file('output/world.gpkg')
```

Note, that if you try to write to the same data source again, the function will overwrite the file.

```
world.to_file('output/world.gpkg')
```

Instead of overwriting the file, we could add new rows to the file with `mode='a'` ('append' mode, as opposed to the default `mode='w'` for the 'write' mode). Appending is supported by several spatial formats, including GeoPackage.

```
world.to_file('output/w_many_features.gpkg')
world.to_file('output/w_many_features.gpkg', mode='a')
```

Now, `w_many_features.gpkg` contains a polygonal layer named `world` with two 'copies' of each country (that is 177×2=354 features, whereas the `world` layer has 177 features).

```
gpd.read_file('output/w_many_features.gpkg').shape
```

```
(354, 11)
```

Alternatively, you can create another, separate, layer, within the same file, which is supported by some formats, including GeoPackage.

```
world.to_file('output/w_many_layers.gpkg')
world.to_file('output/w_many_layers.gpkg', layer='world2')
```

In this case, `w_many_layers.gpkg` has two 'layers': `w_many_layers` (same as the file name, when `layer` is unspecified) and `world2`. Incidentally, the contents of the two layers are identical, but this does not have to be so. Each layer from such a file can be imported separately using the `layer` argument of `gpd.read_file`.

```
layer1 = gpd.read_file('output/w_many_layers.gpkg', layer='w_many_layers')
layer2 = gpd.read_file('output/w_many_layers.gpkg', layer='world2')
```

7.6.2 Raster data

To write a raster file using **rasterio**, we need to pass a raster file path to `rasterio.open` in writing (`'w'`) mode. This implies creating a new empty file (or overwriting an existing one). Next, we need to write the raster values to the file using the `.write` method of the file connection. Finally, we should close the file connection using the `.close` method.

As opposed to reading mode (`'r'`, the default) mode, the `rasterio.open` function in writing mode needs quite a lot of information, in addition to the file path and mode:

- `driver`—The file format. The general recommendation is `'GTiff'` for GeoTIFF, but other formats are also supported (see Table 7.1)
- `height`—Number of rows
- `width`—Number of columns
- `count`—Number of bands
- `nodata`—The value which represents 'No Data', if any
- `dtype`—The raster data type, one of **numpy** types supported by the `driver` (e.g., `np.int64`) (see Table 7.2)
- `crs`—The CRS, e.g., using an EPSG code (such as `4326`)
- `transform`—The transform matrix
- `compress`—A compression method to apply, such as `'lzw'`. This is optional and most useful for large rasters. Note that, at the time of writing, this does not work well[18] for writing multiband rasters

> **Note**
>
> Note that `'GTiff` (GeoTIFF, `.tif`), which is the recommended driver, supports just some of the possible **numpy** data types (see Table 7.2). Importantly, it does not support `np.int64`, the default `int` type. The recommendation in such case it to use `np.int32` (if the range is sufficient), or `np.float64`.

Once the file connection with the right metadata is ready, we do the actual writing using the `.write` method of the file connection. If there are several bands we may execute the `.write` method several times, as in `.write(a,n)`, where `a` is a two-dimensional array representing a single band, and `n` is the band index (starting from `1`, see below). Alternatively, we can write all bands at once, as in `.write(a)`, where `a` is a three-dimensional array. When done, we close the file connection using the `.close` method. Some functions, such as `rasterio.warp.reproject` used for resampling and reprojecting (Section 4.3.3 and Section 6.8) directly accept a file connection in `'w'` mode, thus handling the writing (of a resampled or reprojected raster) for us.

[18] https://gis.stackexchange.com/questions/404738/why-does-rasterio-compression-reduces-image-size-with-single-band-but-not-with-m

Most of the properties are either straightforward to choose, based on our aims (e.g., `driver`, `crs`, `compress`, `nodata`), or directly derived from the array with the raster values itself (e.g., `height`, `width`, `count`, `dtype`). The most complicated property is the `transform`, which specifies the raster origin and resolution. The `transform` is typically either obtained from an existing raster (serving as a 'template'), created from scratch based on manually specified origin and resolution values (e.g., using `rasterio.transform.from_origin`), or calculated automatically (e.g., using `rasterio.warp.calculate_default_transform`), as shown in previous chapters.

Earlier in the book, we have already demonstrated five common scenarios of writing rasters, covering the above-mentioned considerations:

- Creating from scratch (Section 1.3.2)—we created and wrote two rasters from scratch by associating the `elev` and `grain` arrays with an arbitrary spatial extent. The custom arbitrary transformation matrix was created using `rasterio.transform.from_origin`
- Aggregating (Section 4.3.2)—we wrote an aggregated raster, by resampling from an existing raster file, then updating the transformation matrix using `.transform.scale`
- Resampling (Section 4.3.3)—we resampled a raster into a custom grid, manually creating the transformation matrix using `rasterio.transform.from_origin`, then resampling and writing the output using `rasterio.warp.reproject`
- Masking and cropping (Section 5.2)—we wrote masked and/or cropped arrays from a raster, possibly updating the transformation matrix and dimensions (when cropping)
- Reprojecting (Section 6.8)—we reprojected a raster into another CRS, by automatically calculating an optimal `transform` using `rasterio.warp.calculate_default_transform`, then resampling and writing the output using `rasterio.warp.reproject`

To summarize, the raster-writing scenarios differ in two aspects:

1. The way that the transformation matrix for the output raster is obtained:
   - Imported from an existing raster (see below)
   - Created from scratch, using `rasterio.transform.from_origin` (Section 1.3.2)
   - Calculated automatically, using `rasterio.warp.calculate_default_transform` (Section 6.8)
2. The way that the raster is written:
   - Using the `.write` method, given an existing array (Section 1.3.2, Section 4.3.2)
   - Using `rasterio.warp.reproject` to calculate and write a resampled or reprojected array (Section 4.3.3, Section 6.8)

A minimal example of writing a raster file named `r.tif` from scratch, to remind the main concepts, is given below. First, we create a small 2×2 array.

```
r = np.array([1,2,3,4]).reshape(2,2).astype(np.int8)
r
```

```
array([[1, 2],
       [3, 4]], dtype=int8)
```

Next, we define a transformation matrix, specifying the origin and resolution.

```
new_transform = rasterio.transform.from_origin(
    west=-0.5,
    north=51.5,
    xsize=2,
    ysize=2
)
new_transform
```

```
Affine(2.0, 0.0, -0.5,
       0.0, -2.0, 51.5)
```

Then, we establish the writing-mode file connection to `r.tif`, which will be either created or overwritten.

```
dst = rasterio.open(
    'output/r.tif', 'w',
    driver = 'GTiff',
    height = r.shape[0],
    width = r.shape[1],
    count = 1,
    dtype = r.dtype,
    crs = 4326,
    transform = new_transform
)
dst
```

```
<open DatasetWriter name='output/r.tif' mode='w'>
```

Next, we write the array of values into the file connection with the `.write` method. Keep in mind that `r` here is a two-dimensional array representing one band, and `1` is the band index where the array is written into the file.

```
dst.write(r, 1)
```

Finally, we close the connection.

```
dst.close()
```

These expressions, taken together, create a new file `output/r.tif`, which is a 2×2 raster, having a 2 decimal degree resolution, with the top-left corner placed over London.

To make the picture of raster export complete, there are three important concepts we have not covered yet: array and raster data types, writing multiband rasters, and handling 'No Data' values.

Arrays (i.e., `ndarray` objects defined in package **numpy**) are used to store raster values when reading them from file, using `.read` (Section 1.3.1). All values in an array are of the same type, whereas the **numpy** package supports numerous numeric data types of various precision (and, accordingly, memory footprint). Raster formats, such as GeoTIFF, support (a subset of) exactly the same data types as **numpy**, which means that reading a raster file uses as little RAM as possible. The most useful types for raster data, and their support in GeoTIFF are summarized in Table 7.2.

Table 7.2: Commonly used **numpy** data types for rasters, and whether they are supported by the GeoTIFF (`'GTiff'`) file format

Data type	Description	GeoTIFF
`int8`	Integer in a single byte (`-128` to `127`)	
`int16`	Integer in 16 bits (`-32768` to `32767`)	+
`int32`	Integer in 32 bits (`-2147483648` to `2147483647`)	+
`int64`	Integer in 64 bits (`-9223372036854775808` to `9223372036854775807`)	
`uint8`	Unsigned integer in 8 bits (`0` to `255`)	+
`uint16`	Unsigned integer in 16 bits (`0` to `65535`)	+
`uint32`	Unsigned integer in 32 bits (`0` to `4294967295`)	+
`uint64`	Unsigned integer in 64 bits (`0` to `18446744073709551615`)	
`float16`	Half-precision (16 bit) float (`-65504` to `65504`)	
`float32`	Single-precision (32 bit) float (`1e-38` to `1e38`)	+
`float64`	Double-precision (64 bit) float (`1e-308` to `1e308`)	+

The raster data type needs to be specified when writing a raster, typically using the same type as that of the array to be written (e.g., see the `dtype=r.dtype` part in the last example). For an existing raster file, the data type can be queried through the `.dtype` property of the metadata (`.meta['dtype']`).

```
rasterio.open('output/r.tif').meta['dtype']
```

```
'int8'
```

The above expression shows that the GeoTIFF file `r.tif` has the data type `np.int8`, as specified when creating the file with `rasterio.open`, according to the data type of the array we wrote into the file (`dtype=r.dtype`).

```
r.dtype
```

```
dtype('int8')
```

When reading the raster file back into the Python session, the exact same array is recreated.

```
rasterio.open('output/r.tif').read().dtype
```

```
dtype('int8')
```

These code sections demonstrate the agreement between GeoTIFF (and other file formats) data types, which are universal and understood by many programs and programming languages, and the corresponding `ndarray` data types which are defined by **numpy** (Table 7.2).

Writing multiband rasters is similar to writing single-band rasters, only that we need to:

- Define a number of bands other than `count=1`, according to the number of bands we are going to write
- Execute the `.write` method multiple times, once for each layer

For completeness, let's demonstrate writing a multi-band raster named `r3.tif`, which is similar to `r.tif`, but having three bands with values `r*1`, `r*2`, and `r*3` (i.e., the array `r` multiplied by `1`, `2`, or `3`). Since most of the metadata is going to be the same, this is also a good opportunity to (re-)demonstrate updating an existing metadata object rather than creating one from scratch. First, let's make a copy of the metadata we already have in `r.tif`.

```
dst_kwds = rasterio.open('output/r.tif').meta
dst_kwds
```

```
{'driver': 'GTiff',
 'dtype': 'int8',
 'nodata': None,
 'width': 2,
 'height': 2,
 'count': 1,
 'crs': CRS.from_epsg(4326),
 'transform': Affine(2.0, 0.0, -0.5,
        0.0, -2.0, 51.5)}
```

Second, we update the `count` entry, replacing `1` (single-band) with `3` (three-band) using the `.update` method.

```
dst_kwds.update(count=3)
dst_kwds
```

```
{'driver': 'GTiff',
 'dtype': 'int8',
 'nodata': None,
 'width': 2,
 'height': 2,
 'count': 3,
 'crs': CRS.from_epsg(4326),
 'transform': Affine(2.0, 0.0, -0.5,
        0.0, -2.0, 51.5)}
```

Finally, we can create a file connection using the updated metadata, write the values of the three bands, and close the connection (note that we are switching to the 'keyword argument' syntax of Python function calls here; see note in Section 4.3.2).

```
dst = rasterio.open('output/r3.tif', 'w', **dst_kwds)
dst.write(r*1, 1)
dst.write(r*2, 2)
dst.write(r*3, 3)
dst.close()
```

As a result, a three-band raster named `r3.tif` is created.

Rasters often contain 'No Data' values, representing missing data, for example, unreliable measurements due to clouds or pixels outside of the photographed extent. In a **numpy** `ndarray` object, 'No Data' values may be represented by the special `np.nan` value. However, due to computer memory limitations, only arrays of type `float` can contain `np.nan`, while arrays of type `int` cannot. For `int` rasters containing 'No Data', we typically mark missing data with a specific value beyond the valid range (e.g., `-9999`). The missing data 'flag' definition is stored in the file (set through the `nodata` property of the file connection, see above). When reading an `int` raster with 'No Data' back into Python, we need to be aware of the flag, if any. Let's demonstrate it through examples.

We will start with the simpler case, rasters of type `float`. Since `float` arrays may contain the 'native' value `np.nan`, representing 'No Data' is straightforward. For example, suppose that we have a `float` array of size 2×2 containing one `np.nan` value.

```
r = np.array([1.1,2.1,np.nan,4.1]).reshape(2,2)
r
```

```
array([[1.1, 2.1],
       [nan, 4.1]])
```

```
r.dtype
```

```
dtype('float64')
```

When writing this type of array to a raster file, we do not need to specify any particular `nodata` 'flag' value.

```
dst = rasterio.open(
    'output/r_nodata_float.tif', 'w',
    driver = 'GTiff',
    height = r.shape[0],
    width = r.shape[1],
    count = 1,
    dtype = r.dtype,
    crs = 4326,
    transform = new_transform
)
dst.write(r, 1)
dst.close()
```

This is equivalent to `nodata=None`.

```
rasterio.open('output/r_nodata_float.tif').meta
```

```
{'driver': 'GTiff',
 'dtype': 'float64',
 'nodata': None,
 'width': 2,
 'height': 2,
 'count': 1,
 'crs': CRS.from_epsg(4326),
 'transform': Affine(2.0, 0.0, -0.5,
        0.0, -2.0, 51.5)}
```

Reading from the raster back into the Python session reproduces the same exact array, including `np.nan`.

```
rasterio.open('output/r_nodata_float.tif').read()
```

```
array([[[1.1, 2.1],
        [nan, 4.1]]])
```

Now, conversely, suppose that we have an `int` array with missing data, where the 'missing' value must inevitably be marked using a specific `int` 'flag' value, such as `-9999` (remember that we can't store `np.nan` in an `int` array!).

```
r = np.array([1,2,-9999,4]).reshape(2,2).astype(np.int32)
r
```

```
array([[    1,     2],
       [-9999,     4]], dtype=int32)
```

```
r.dtype
```

```
dtype('int32')
```

When writing the array to file, we must specify `nodata=-9999` to keep track of our 'No Data' flag.

```
dst = rasterio.open(
    'output/r_nodata_int.tif', 'w',
    driver = 'GTiff',
    height = r.shape[0],
    width = r.shape[1],
    count = 1,
    dtype = r.dtype,
    nodata = -9999,
    crs = 4326,
    transform = new_transform
)
dst.write(r, 1)
dst.close()
```

Examining the metadata of the file we've just created confirms that the `nodata=-9999` setting was stored in the file `r_nodata_int.tif`.

```
rasterio.open('output/r_nodata_int.tif').meta
```

```
{'driver': 'GTiff',
 'dtype': 'int32',
 'nodata': -9999.0,
 'width': 2,
 'height': 2,
 'count': 1,
 'crs': CRS.from_epsg(4326),
 'transform': Affine(2.0, 0.0, -0.5,
        0.0, -2.0, 51.5)}
```

If you try to open the file in GIS software, such as QGIS, you will see the missing data interpreted (e.g., the pixel shown as blank), meaning that the software is aware of the flag. However, reading the data back into Python reproduces an `int` array with `-9999`, due to the limitation of `int` arrays stated before.

```
src = rasterio.open('output/r_nodata_int.tif')
r = src.read()
r
```

```
array([[[    1,     2],
        [-9999,     4]]], dtype=int32)
```

The Python user must therefore be mindful of 'No Data' `int` rasters, for example to avoid interpreting the value `-9999` literally. For instance, if we 'forget' about the `nodata` flag, the literal calculation of the `.mean` would incorrectly include the value `-9999`.

```
r.mean()
```

```
np.float64(-2498.0)
```

There are two basic ways to deal with the situation: either converting the raster to `float`, or using a 'No Data' mask. The first approach, simple and particularly relevant for small rasters where memory constraints are irrelevant, is to go from `int` to `float`, to gain the ability of the natural `np.nan` representation. Here is how we can do this with `r_nodata_int.tif`. We detect the missing data flag, convert the raster to `float`, then assign `np.nan` into the cells that are supposed to be missing.

```
mask = r == src.nodata
r = r.astype(np.float64)
r[mask] = np.nan
r
```

```
array([[[ 1.,  2.],
        [nan,  4.]]])
```

From there on, we deal with `np.nan` the usual way, such as using `np.nanmean` to calculate the mean excluding 'No Data'.

```
np.nanmean(r)
```

```
np.float64(2.3333333333333335)
```

The second approach is to read the values into a so-called *'masked' array*, using the argument `masked=True` of the `.read` method. A masked array can be thought of as an extended `ndarray`, with two components: `.data` (the values) and `.mask` (a corresponding boolean array marking 'No Data' values).

```
r = src.read(masked=True)
r
```

```
masked_array(
  data=[[[1, 2],
         [--, 4]]],
  mask=[[[False, False],
         [ True, False]]],
  fill_value=-9999,
  dtype=int32)
```

Complete treatment of masked arrays is beyond the scope of this book. However, the basic idea is that many **numpy** operations 'honor' the mask, so that the

user does not have to keep track of the way that 'No Data' values are marked, similarly to the natural `np.nan` representation and regardless of the data type. For example, the `.mean` of a masked array ignores the value `-9999`, because it is masked, taking into account just the valid values `1`, `2`, and `4`.

```
r.mean()
```

```
np.float64(2.3333333333333335)
```

Switching to `float` and assigning `np.nan` is the simpler approach, since that way we can keep working with the familiar `ndarray` data structure for all raster types, whether `int` or `float`. Nevertheless, learning how to work with masked arrays can be beneficial when we have good reasons to keep our raster data in `int` arrays (for example, due to RAM limits) and still perform operations that take missing values into account.

Finally, keep in mind that, confusingly, `float` rasters may represent 'No Data' using a specific 'flag' (such as `-9999.0`), instead, or in addition to (!), the native `np.nan` representation. In such cases, the same considerations shown for `int` apply to `float` rasters as well.

8

Making maps with Python

Prerequisites

This chapter requires importing the following packages:

```
import matplotlib.pyplot as plt
import geopandas as gpd
import rasterio
import rasterio.plot
import contextily as cx
import folium
```

It also relies on the following data files:

```
nz = gpd.read_file('data/nz.gpkg')
nz_height = gpd.read_file('data/nz_height.gpkg')
nz_elev = rasterio.open('data/nz_elev.tif')
```

8.1 Introduction

A satisfying and important aspect of geographic research is communicating the results. Map making—the art of cartography—is an ancient skill that involves communication, intuition, and an element of creativity. In addition to being fun and creative, cartography also has important practical applications. A carefully crafted map can be the best way of communicating the results of your work, but poorly designed maps can leave a bad impression. Common design issues include poor placement, size, and readability of text and careless selection of colors, as outlined in the style guide of the Journal of Maps. Furthermore, poor map making can hinder the communication of results (Brewer 2015):

> Amateur-looking maps can undermine your audience's ability to understand important information and weaken the presentation of a professional data investigation.

DOI: 10.1201/9781003379911-8

Maps have been used for several thousand years for a wide variety of purposes. Historic examples include maps of buildings and land ownership in the Old Babylonian dynasty more than 3000 years ago and Ptolemy's world map in his masterpiece Geography nearly 2000 years ago (Talbert 2014).

Map making has historically been an activity undertaken only by, or on behalf of, the elite. This has changed with the emergence of open-source mapping software such as mapping packages in Python, R, and other languages, and the 'print composer' in QGIS, which enable anyone to make high-quality maps, enabling 'citizen science'. Maps are also often the best way to present the findings of geocomputational research in a way that is accessible. Map making is therefore a critical part of geocomputation and its emphasis not only on describing, but also changing the world.

Basic static display of vector layers in Python is done with the `.plot` method or the `rasterio.plot.show` function, for vector layers and rasters, as we saw in Section 1.2.2 and Section 1.3.1, respectively. Other, more advanced uses of these methods, were also encountered in subsequent chapters, when demonstrating the various outputs we got. In this chapter, we provide a comprehensive summary of the most useful workflows of these two methods for creating static maps (Section 8.2). Static maps can be easily shared and viewed (whether digitally or in print), however they can only convey as much information as a static image can. Interactive maps provide much more flexibility in terms of user experience and amount of information, however they often require more work to design and effectively share. Thus, in Section 8.3, we move on to elaborate on the `.explore` method for creating interactive maps, which was also briefly introduced earlier in Section 1.2.2.

8.2 Static maps

Static maps are the most common type of visual output from geocomputation. For example, we have been using `.plot` and `rasterio.plot.show` throughout the book, to display **geopandas** and **rasterio** geocomputation results, respectively. In this section, we systematically review and elaborate on the various properties that can be customized when using those functions.

A static map is basically a digital image. When stored in a file, standard formats include `.png` and `.pdf` for graphical raster and vector outputs, respectively. Thanks to their simplicity, static maps can be shared in a wide variety of ways: in print, through files sent by e-mail, embedded in documents and web pages, etc.

Nevertheless, there are many aesthetic considerations when making a static map, and there is also a wide variety of ways to create static maps using novel

presentation methods. This is the focus of the field of cartography, and beyond the scope of this book.

Let's move on to the basics of static mapping with Python.

8.2.1 Minimal examples

A vector layer (`GeoDataFrame`) or a geometry column (`GeoSeries`) can be displayed using their `.plot` method (Section 1.2.2). A minimal example of a vector layer map is obtained using `.plot` with nothing but the defaults (Figure 8.1).

```
nz.plot();
```

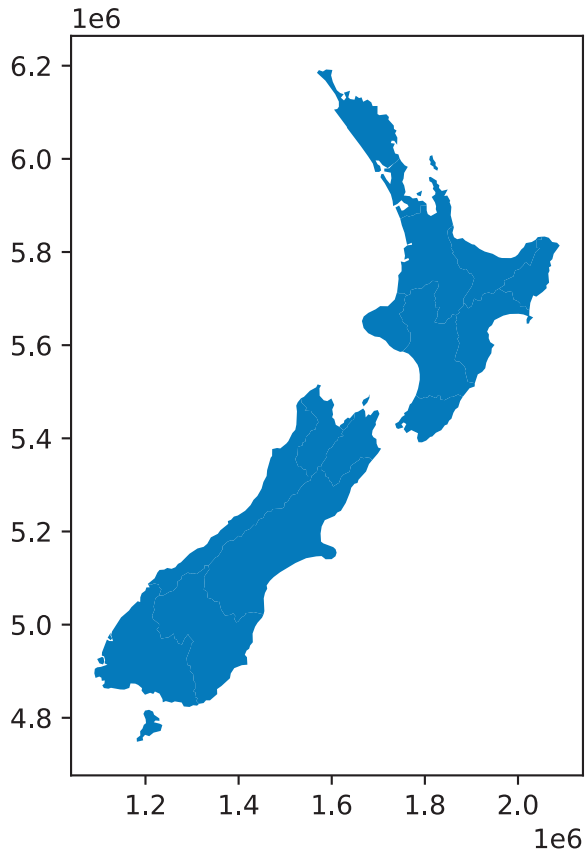


Figure 8.1: Minimal example of a static vector layer plot with `.plot`

A `rasterio` raster file connection, or a numpy `ndarray`, can be displayed using `rasterio.plot.show` (Section 1.3.1). Figure 8.2 shows a minimal example of a static raster map.

```
rasterio.plot.show(nz_elev);
```

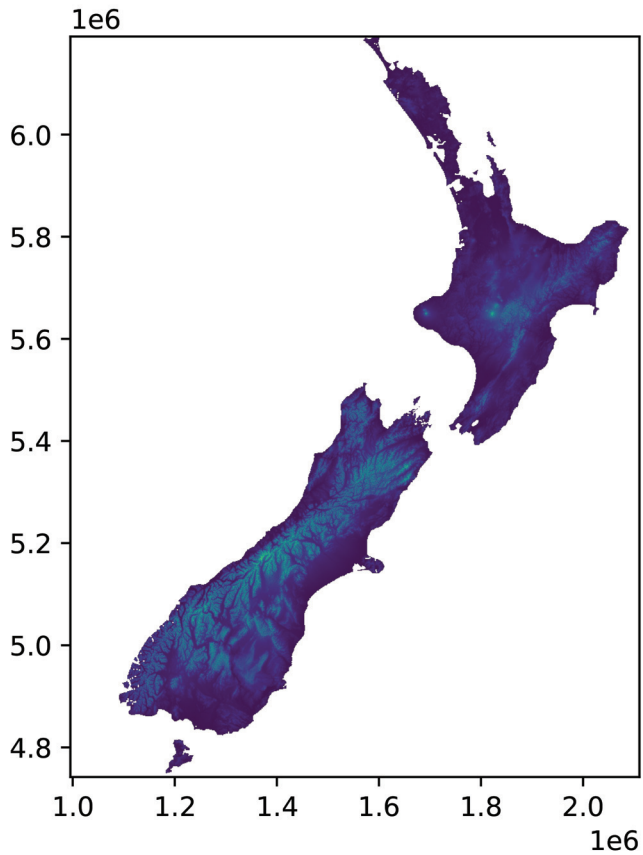
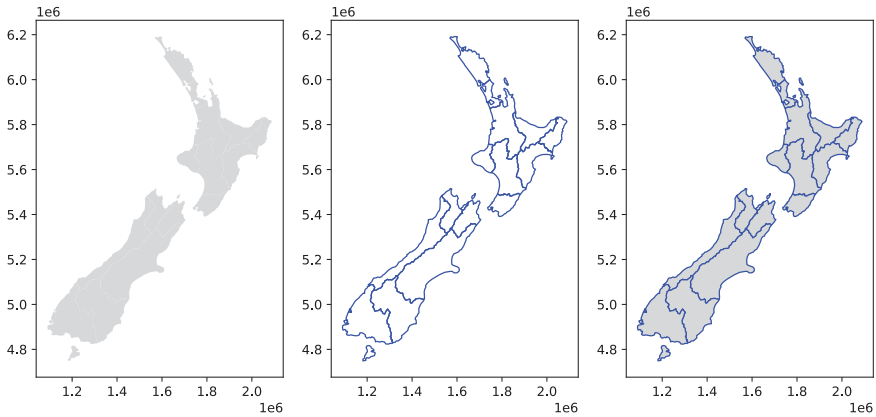


Figure 8.2: Minimal example of a static raster plot with `rasterio.plot.show`

8.2.2 Styling

The most useful visual properties of the geometries, that can be specified in `.plot`, include `color`, `edgecolor`, and `markersize` (for points) (Figure 8.3).

```
nz.plot(color='lightgrey');
nz.plot(color='none', edgecolor='blue');
nz.plot(color='lightgrey', edgecolor='blue');
```



(a) Light grey fill (b) No fill, blue edge (c) Light grey fill, blue edge

Figure 8.3: Setting `color` and `edgecolor` in static maps of a vector layer

The next example uses `markersize` to get larger points (Figure 8.4). It also demonstrates how to control the overall figure size, such as 4×4 *in* in this

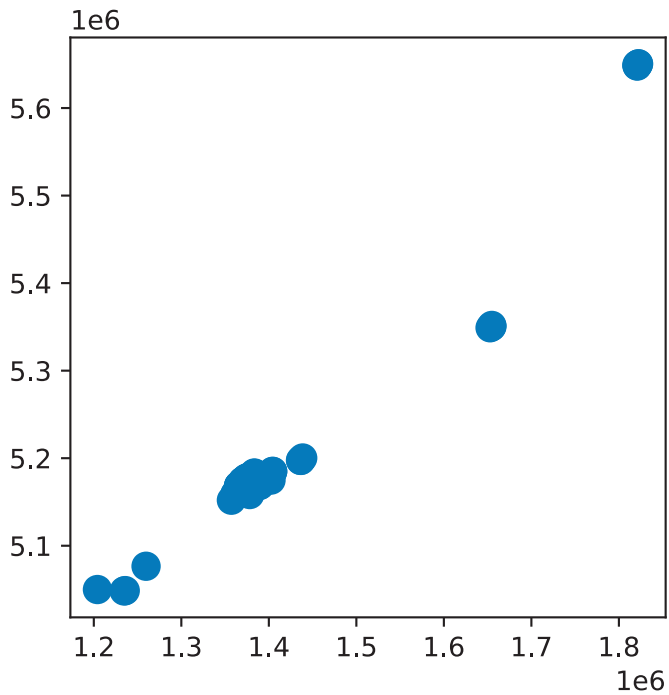


Figure 8.4: Setting `markersize` in a static map of a vector layer

case, using `plt.subplots` to initialize the plot and its `figsize` parameter to specify dimensions.

```
fig, ax = plt.subplots(figsize=(4,4))
nz_height.plot(markersize=100, ax=ax);
```

> **Note**
>
> As you have probably noticed throughout the book, the `plt.subplots` function is used to initialize a **maptplotlib** plot layout, possibly also specifying image size (e.g., Figure 8.4) and multi-panel layout (e.g., Figure 8.18). The returned value is a `tuple` of `Figure` and `Axes` objects, which we conventionally unpack to variables named `fig` and `ax`. These two variables represent the entire figure, and the elements of individual sub-figures, respectively.
> For our purposes in this book, we have been using just the `ax` object, passing it to the `ax` parameter in further function calls, in order to add subsequent layers (e.g., Figure 8.16) or other elements (e.g., Figure 8.10) into the same panel. In a single-panel figure, we pass `ax` itself, whereas in a multi-panel figure we pass individual elements representing a specific panel (such as `ax[0]` or `ax[0][0]`, depending of the layout; see Section 8.2.7)
> Note that in some of the cases we have used an alternative to `plt.subplots`—we assigned an initial plot into a variable, conventionally named `base`, similarly passing it to the `ax` parameter of further calls, e.g., to add subsequent layers (e.g., Figure 8.14); this (shorter) syntax, though, is less general than `plt.subplots` and not applicable in some of the cases (such as displaying a raster and a vector layer in the same plot, e.g., Figure 8.16).

8.2.3 Symbology

We can set symbology in a `.plot` using the following parameters:

- `column`—a column name
- `legend`—whether to show a legend
- `cmap`—color map, a.k.a. color scale, a palette from which the colors are sampled

For example, Figure 8.5 shows the `nz` polygons colored according to the `'Median_income'` attribute (column), with a legend.

```
nz.plot(column='Median_income', legend=True);
```

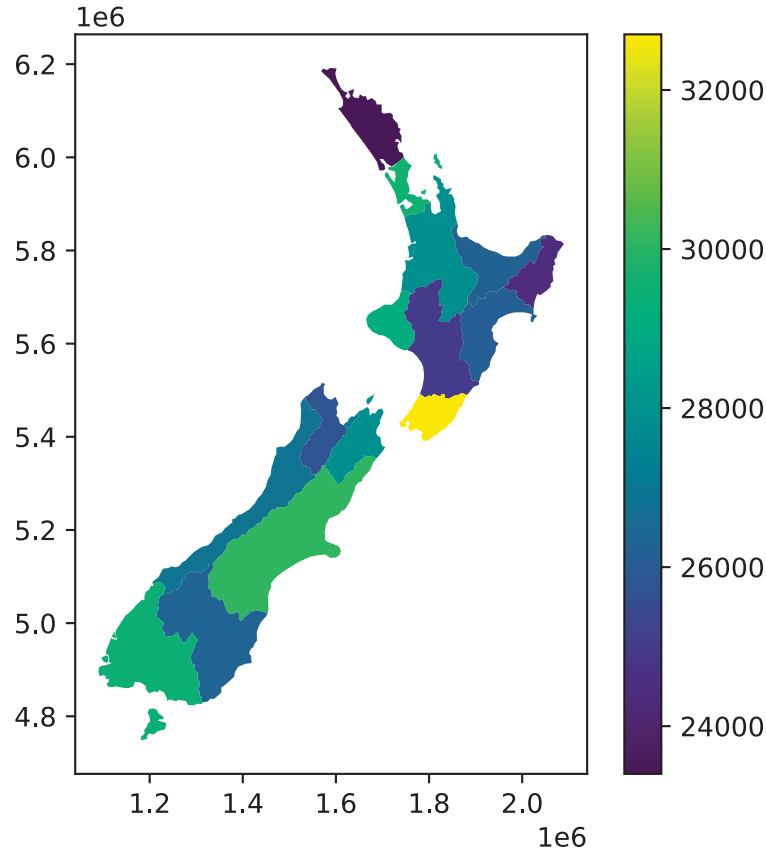
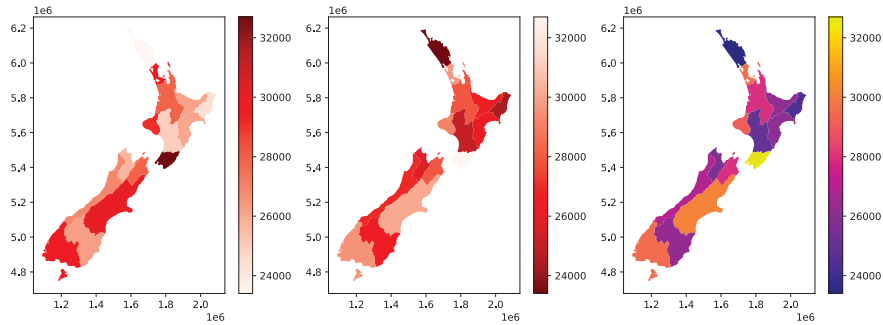


Figure 8.5: Symbology in a static map created with `.plot`

The default color scale which you see in Figure 8.5 is `cmap='viridis'`. The `cmap` ('color map') argument can be used to specify one of countless color scales. A first safe choice is often the ColorBrewer[1] collection of color scales, specifically designed for mapping. Any color scale can be reversed, using the `_r` suffix. Finally, other color scales are available: see the **matplotlib** colormaps article[2] for details. The following code section demonstrates three-color scale specifications other than the default (Figure 8.6).

[1] https://colorbrewer2.org/
[2] https://matplotlib.org/stable/tutorials/colors/colormaps.html

```
nz.plot(column='Median_income', legend=True, cmap='Reds');
nz.plot(column='Median_income', legend=True, cmap='Reds_r');
nz.plot(column='Median_income', legend=True, cmap='plasma');
```



(a) The `'Reds'` color scale from ColorBrewer

(b) Reversed `'Reds'` color scale

(c) The `'plasma'` color scale from **matplotlib**

Figure 8.6: Symbology in a static map of a vector layer, created with `.plot`

Categorical symbology is also supported, such as when `column` points to an `str` attribute. For categorical variables, it makes sense to use a qualitative color scale, such as `'Set1'` from ColorBrewer. For example, the following expression sets symbology according to the `'Island'` column (Figure 8.7).

```
nz.plot(column='Island', legend=True, cmap='Set1');
```

In case the legend interferes with the contents (such as in Figure 8.7), we can modify the legend position using the `legend_kwds` argument (Figure 8.8).

```
nz.plot(column='Island', legend=True, cmap='Set1', legend_kwds={'loc': 4});
```

The `rasterio.plot.show` function is also based on **matplotlib** (Hunter 2007), and thus supports the same kinds of `cmap` arguments (Figure 8.9).

```
rasterio.plot.show(nz_elev, cmap='BrBG');
rasterio.plot.show(nz_elev, cmap='BrBG_r');
rasterio.plot.show(nz_elev, cmap='gist_earth');
```

Unfortunately, there is no built-in option to display a legend in `rasterio.plot.show`. The following workaround, reverting to **matplotlib** methods, can be used to achieve it instead (Figure 8.10). Basically, the code reverts to the **matplotlib** `.colorbar` method to add a legend, using the `plt.imshow` function that draws an image of a **numpy** array (which `rasterio.plot.show` is a wrapper of).

```
fig, ax = plt.subplots()
i = plt.imshow(nz_elev.read(1), cmap='BrBG')
```

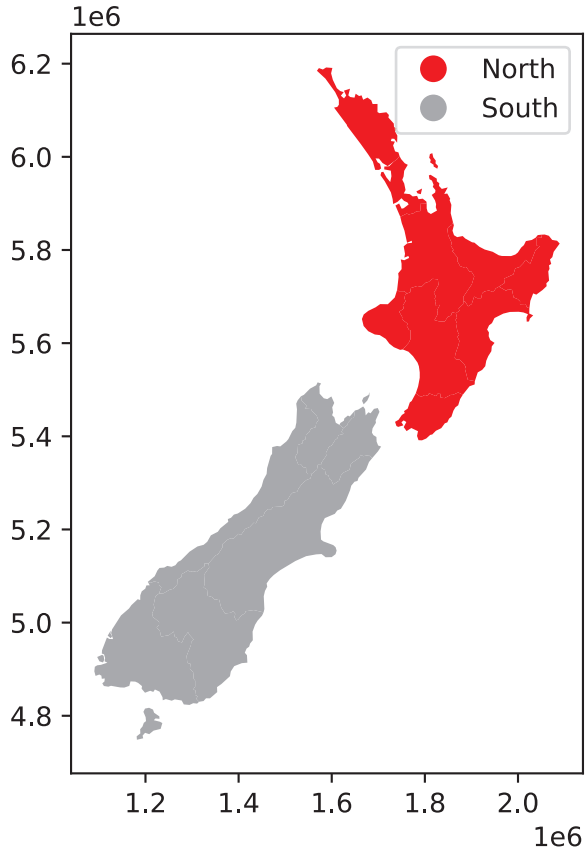


Figure 8.7: Symbology for a categorical variable

```
rasterio.plot.show(nz_elev, cmap='BrBG', ax=ax);
fig.colorbar(i, ax=ax);
```

8.2.4 Labels

Labels are often useful to annotate maps and identify the location of specific features. GIS software, as opposed to **matplotlib**, has specialized algorithms for label placement, e.g., to avoid overlaps between adjacent labels. Furthermore, editing in graphical editing software is sometimes used for fine-tuning of label placement. Nevertheless, simple labels added within the Python environment can be a good starting point, both for interactive exploration and sharing analysis results.

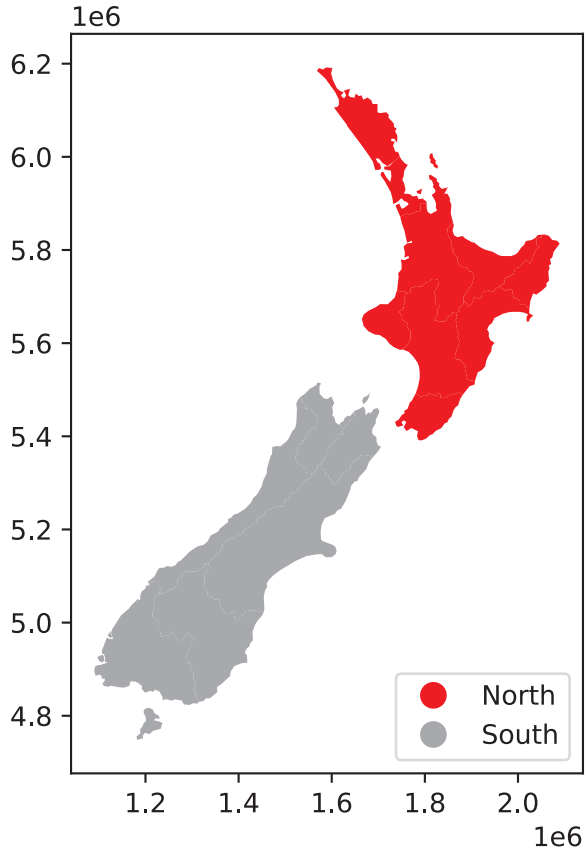
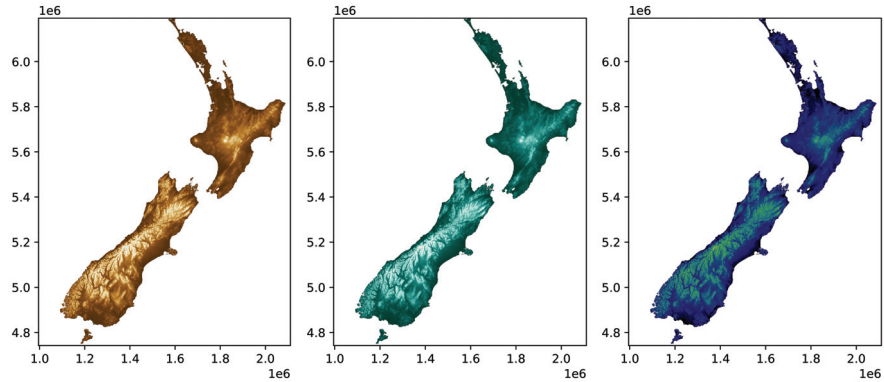


Figure 8.8: Setting legend position in `.plot`

To demonstrate it, suppose that we have a layer `nz1` of regions comprising the New Zealand southern Island.

```
nz1 = nz[nz['Island'] == 'South']
```

To add a label in **matplotlib**, we use the `.annotate` method where the important arguments are the label string and the placement (a `tuple` of the form `(x,y)`). When labeling vector layers, we typically want to add numerous labels, based on (one or more) attribute of each feature. To do that, we can run a `for` loop, or use the `.apply` method, to pass the label text and the coordinates of each feature to `.annotate`. In the following example, we use the `.apply` method the pass the region name (`'Name'` attribute) and the geometry centroid coordinates, for each region, to `.annotate`. We are also using `ha`, short for `horizontalalignment`, with `'center'` (other options are `'right'` and `'left'`) (Figure 8.11).



(a) The `'BrBG'` color scale from ColorBrewer

(b) Reversed `'BrBG_r'` color scale

(c) The `'gist_earth'` color scale from **matplotlib**

Figure 8.9: Symbology in a static map of a raster, with `rasterio.plot.show`

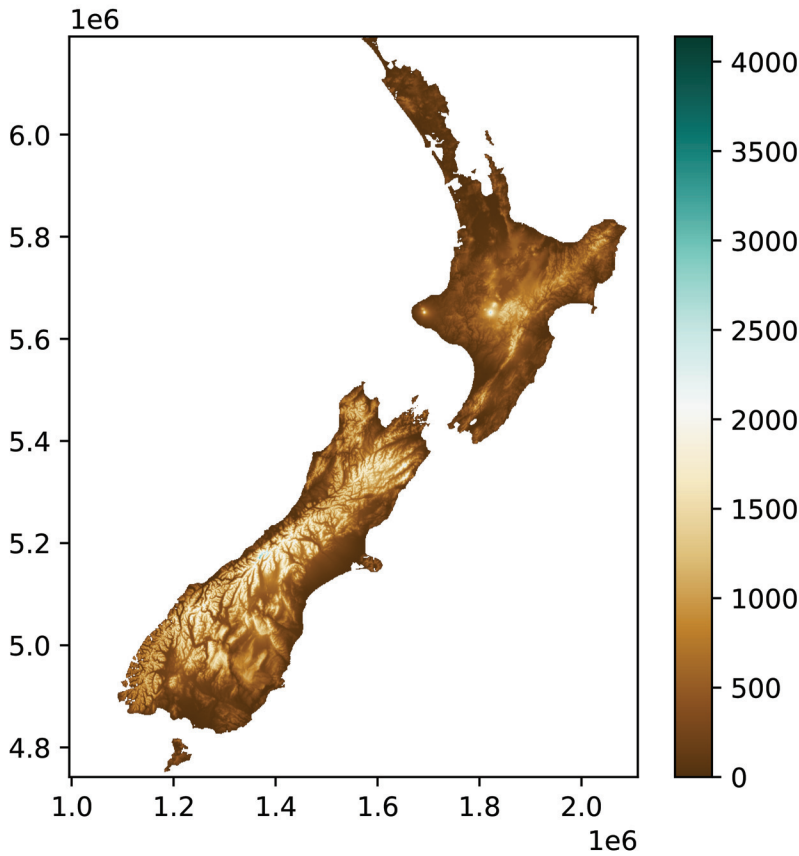


Figure 8.10: Adding a legend in `rasterio.plot.show`

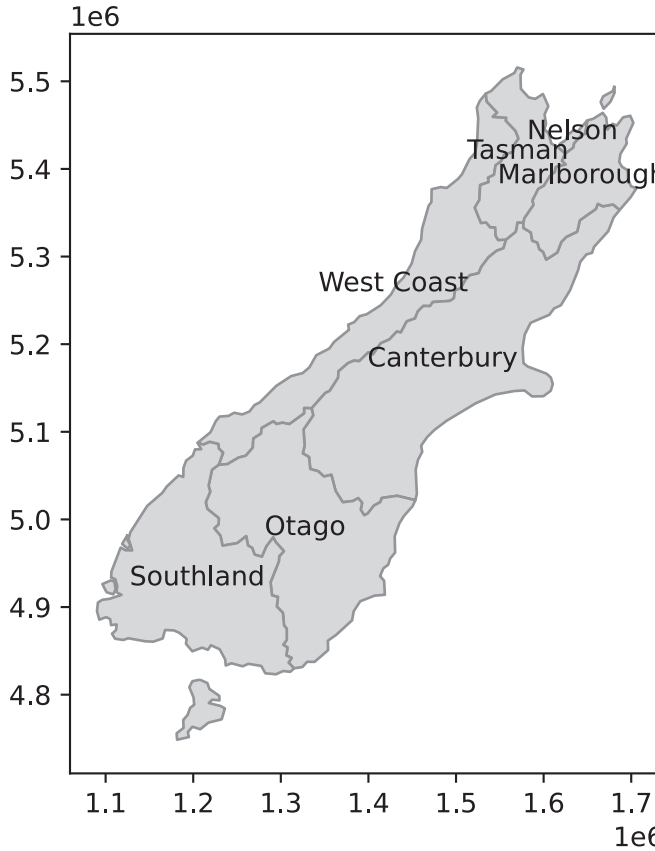


Figure 8.11: Labels at polygon centroids

```
fig, ax = plt.subplots()
nz1.plot(ax=ax, color='lightgrey', edgecolor='grey')
nz1.apply(
    lambda x: ax.annotate(
        text=x['Name'],
        xy=x.geometry.centroid.coords[0],
        ha='center'
    ),
    axis=1
);
```

As another example, let's create a map of all regions of New Zealand, with labels for the island names. First, we will calculate the island centroids, which will be the label placement positions.

```
ctr = nz[['Island', 'geometry']].dissolve(by='Island').reset_index()
ctr['geometry'] = ctr.centroid
ctr
```

	Island	geometry
0	North	POINT (1834096.904 5732233.908)
1	South	POINT (1401304.646 5125013.652)

Then, we again use `.apply`, combined with `.annotate`, to add the text labels. The main difference compared to the previous example (Figure 8.11) is that we are directly passing the geometry coordinates (`.geometry.coords[0]`), since the geometries are points rather than polygons. We are also using the `weight='bold'` argument to use bold font (Figure 8.12).

```
fig, ax = plt.subplots()
nz.plot(ax=ax, color='none', edgecolor='lightgrey')
ctr.apply(
    lambda x: ax.annotate(
        text=x['Island'],
        xy=x.geometry.coords[0],
        ha='center',
        weight='bold'
    ),
    axis=1
);
```

It should be noted that sometimes we wish to add text labels 'manually', one by one, rather than use a loop or `.apply`. For example, we may want to add labels of specific locations not stored in a layer, or to have control over the specific properties of each label. To add text labels manually, we can run the `.annotate` expressions one at a time, as shown in the code section below recreating the last result with the 'manual' approach (Figure 8.13).

```
fig, ax = plt.subplots()
nz.plot(ax=ax, color='none', edgecolor='lightgrey')
ax.annotate('This is label 1', (1.8e6, 5.8e6), ha='center', weight='bold')
ax.annotate('This is label 2', (1.4e6, 5.2e6), ha='center', weight='bold');
```

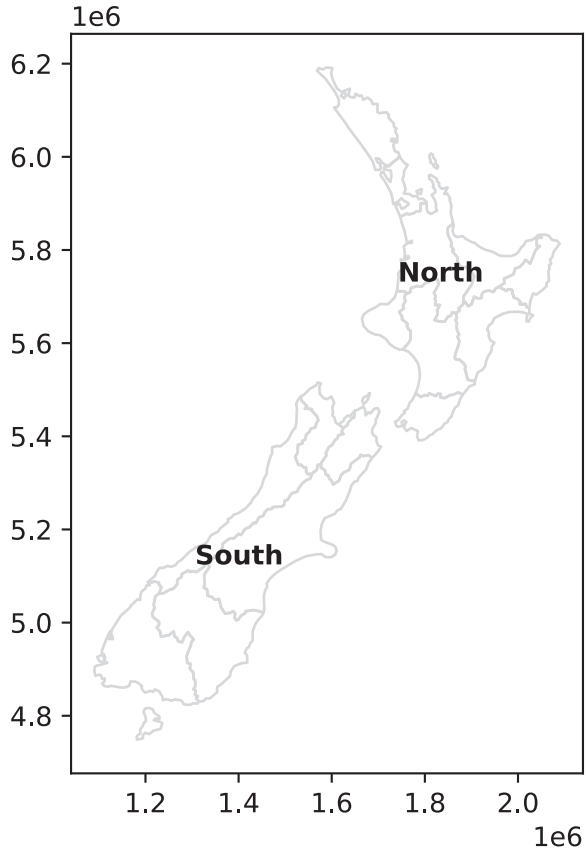


Figure 8.12: Labels at points

8.2.5 Layers

To display more than one layer in the same static map, we can:

1. Store the first plot in a variable (e.g., `base`)
2. Pass it as the `ax` argument of any subsequent plot(s) (e.g., `ax=base`)

For example, here is how we can plot `nz` and `nz_height` together (Figure 8.14).

```
base = nz.plot(color='none')
nz_height.plot(ax=base, color='red');
```

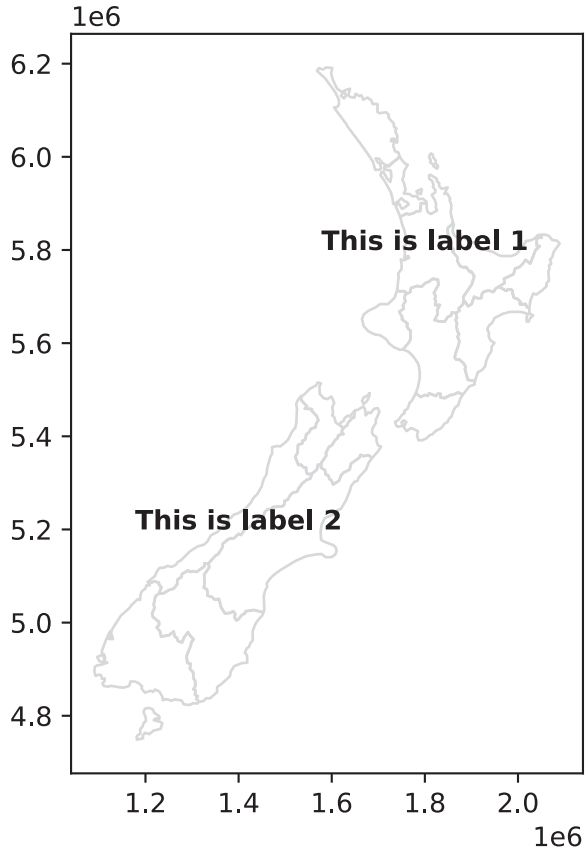


Figure 8.13: Labels at points (manual)

Alternatively (see note in Section 8.2.2), we can:

1. Initialize the plot using `fig,ax=plt.subplots()`
2. Pass `ax` to any subsequent plot

```
fig, ax = plt.subplots()
nz.plot(ax=ax, color='none')
nz_height.plot(ax=ax, color='red');
```

We can combine rasters and vector layers in the same plot as well, which we already did earlier in the book, for example when explaining masking and cropping (Figure 5.2). The technique is to initialize a plot with

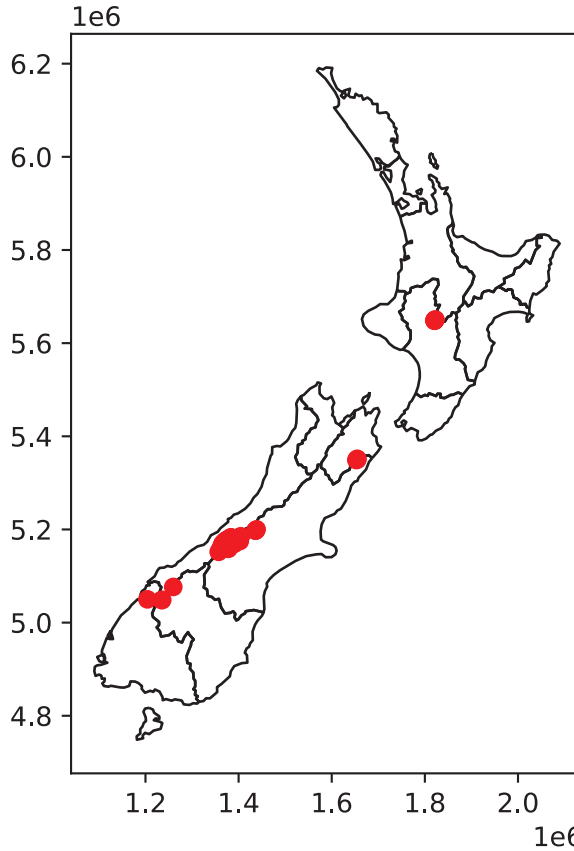


Figure 8.14: Plotting two layers, `nz` (polygons) and `nz_height` (points)

`fig,ax=plt.subplots()`, then pass `ax` to any of the separate plots, making them appear together.

For example, Figure 8.16 demonstrates plotting a raster with increasingly complicated additions:

- Panel (a) shows a raster (New Zealand elevation) and a vector layer (New Zealand administrative division)
- Panel (b) shows the raster with a buffer of 22.2 *km* around the dissolved administrative borders, representing New Zealand's territorial waters (see Section 3.3.6)
- Panel (c) shows the raster with two vector layers: the territorial waters (in red) and elevation measurement points (in yellow)

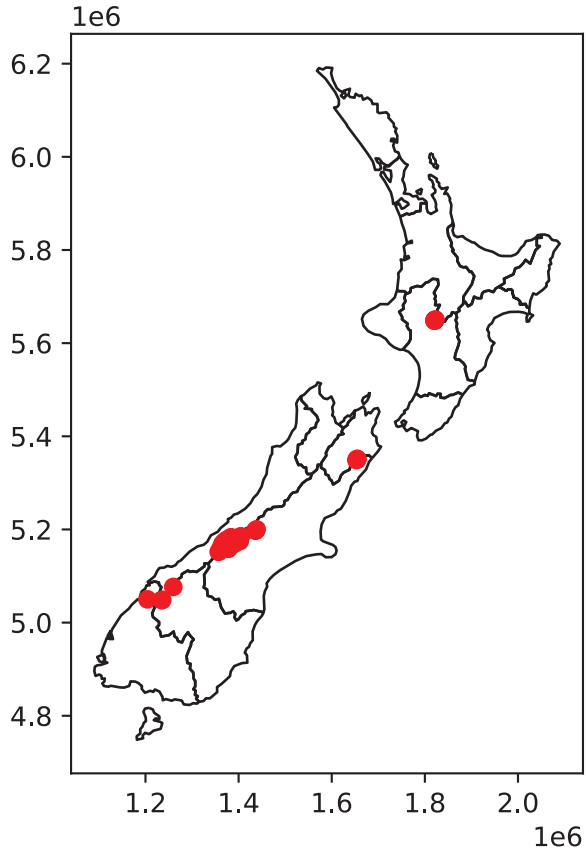
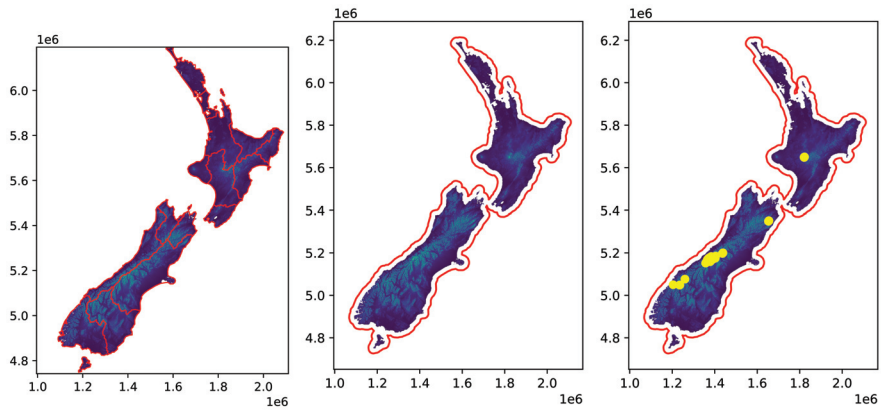


Figure 8.15: Plotting two layers, `nz` (polygons) and `nz_height` (points), using `plt.subplots`

```
# Raster + vector layer
fig, ax = plt.subplots(figsize=(5, 5))
rasterio.plot.show(nz_elev, ax=ax)
nz.to_crs(nz_elev.crs).plot(ax=ax, color='none', edgecolor='red');
# Raster + computed vector layer
fig, ax = plt.subplots(figsize=(5, 5))
rasterio.plot.show(nz_elev, ax=ax)
gpd.GeoSeries(nz.union_all(), crs=nz.crs) \
    .to_crs(nz_elev.crs) \
    .buffer(22200) \
    .exterior \
    .plot(ax=ax, color='red');
# Raster + two vector layers
```

```
fig, ax = plt.subplots(figsize=(5, 5))
rasterio.plot.show(nz_elev, ax=ax)
gpd.GeoSeries(nz.union_all(), crs=nz.crs) \
    .to_crs(nz_elev.crs) \
    .buffer(22200) \
    .exterior \
    .plot(ax=ax, color='red')
nz_height.to_crs(nz_elev.crs).plot(ax=ax, color='yellow');
```



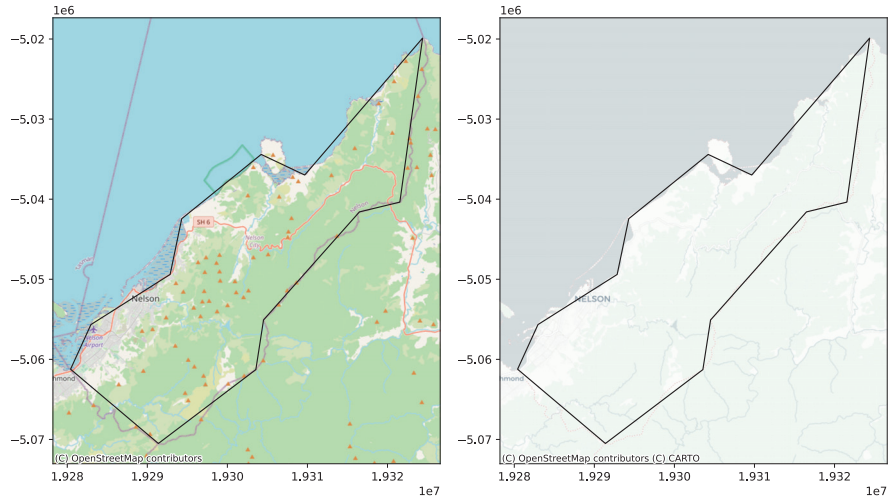
(a) Raster + vector layer (b) Raster + computed vector layer (c) Raster + two vector layers

Figure 8.16: Combining a raster and vector layers in the same plot

> **Note**
>
> Note that the drawing order of layers is not necessarily according to the order of expressions, in the code, but according to layer *type*. For example, by default line layers are drawn on top of point layers. To override the default plotting order, we can use the `zorder` argument of `.plot`. Layers with higher `zorder` values will be drawn on top. For example, the following would draw `layer2` on top of `layer1` (regardless of their types).
>
> ```
> base = layer1.plot(zorder=1)
> layer2.plot(ax=base, zorder=2);
> ```

8.2.6 Basemaps

Basemaps, or background layers, are often useful to provide context to the displayed layers (which are in the 'foreground'). Basemaps are ubiquitous in



(a) 'OpenStreetMap' basemap (b) 'CartoDB Positron' basemap

Figure 8.17: Adding a basemap to a static map, using **contextily**

interactive maps (see Section 8.3). However, they are often useful in static maps too.

Basemaps can be added to **geopandas** static plots using the **contextily** package. A preliminary step is to convert our layers to `EPSG:3857` ('Web Mercator'), to be in agreement with the basemaps, which are typically provided in this CRS[3]. For example, let's take the small `"Nelson"` polygon from `nz`, and reproject it to `3857`.

```
nzw = nz[nz['Name'] == 'Nelson'].to_crs(epsg=3857)
```

To add a basemap, we use the `cx.add_basemap` function, similarly to the way we added multiple layers (Section 8.2.5). The default basemap is 'OpenStreetMap'. You can specify a different basemap using the `source` parameter, with one of the values in `cx.providers` (Figure 8.17).

```
# OpenStreetMap
fig, ax = plt.subplots(figsize=(7, 7))
ax = nzw.plot(color='none', ax=ax)
cx.add_basemap(ax, source=cx.providers.OpenStreetMap.Mapnik);
# CartoDB.Positron
fig, ax = plt.subplots(figsize=(7, 7))
ax = nzw.plot(color='none', ax=ax)
cx.add_basemap(ax, source=cx.providers.CartoDB.Positron);
```

[3]Another option is to reproject the tiles to match the CRS of the foreground layers; this is less commonly used workflow, as it may lead to distorted appearance of the background layer.

Check out the gallery[4] for more possible basemaps. Custom basemaps (such as from your own raster tile server) can be also specified using a URL. Finally, you may read the *Adding a background map to plots*[5] tutorial for more examples.

8.2.7 Faceted maps

Faceted maps are multiple maps displaying the same symbology for the same spatial layers, but with different data in each panel. The data displayed in the different panels typically refer to different properties, or time steps. For example, the `nz` layer has several different properties for each polygon, stored as separate attributes:

```
vars = ['Land_area', 'Population', 'Median_income', 'Sex_ratio']
nz[vars]
```

	Land_area	Population	Median_income	Sex_ratio
0	12500.561149	175500.0	23400	0.942453
1	4941.572557	1657200.0	29600	0.944286
2	23900.036383	460100.0	27900	0.952050
...	...	...	...	...
13	9615.976035	51100.0	25700	0.971898
14	422.195242	51400.0	27200	0.925967
15	10457.745485	46200.0	27900	0.957792

We may want to plot them all in a faceted map, that is, four small maps of `nz` with the different variables. To do that, we initialize the plot with the expected number of panels, such as `ncols=len(vars)` if we wish to have one row and four columns, and then go over the variables in a `for` loop, each time plotting `vars[i]` into the `ax[i]` panel (Figure 8.18).

```
fig, ax = plt.subplots(ncols=len(vars), figsize=(9, 2))
for i in range(len(vars)):
    nz.plot(ax=ax[i], column=vars[i], legend=True)
    ax[i].set_title(vars[i])
```

In case we prefer a specific layout, rather than one row or one column, we can initialize the required number or rows and columns, as in `plt.subplots(nrows,ncols)`, 'flatten' `ax`, so that the facets are still accessible using a single index `ax[i]` (rather than the default `ax[i][j]`), and plot into `ax[i]`. For example, here is how we can reproduce the last plot, this time in a 2×2 layout, instead of a 1×4 layout (Figure 8.19). One more modification we are doing here is hiding the axis ticks and labels, to make the map less 'crowded', using `ax[i].xaxis.set_visible(False)` (and same for `.yaxis`).

[4] https://xyzservices.readthedocs.io/en/stable/gallery.html
[5] https://geopandas.org/en/stable/gallery/plotting_basemap_background.html

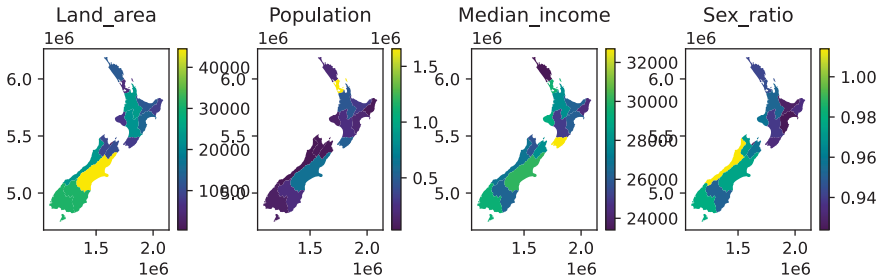


Figure 8.18: Faceted map, four different variables of `nz`

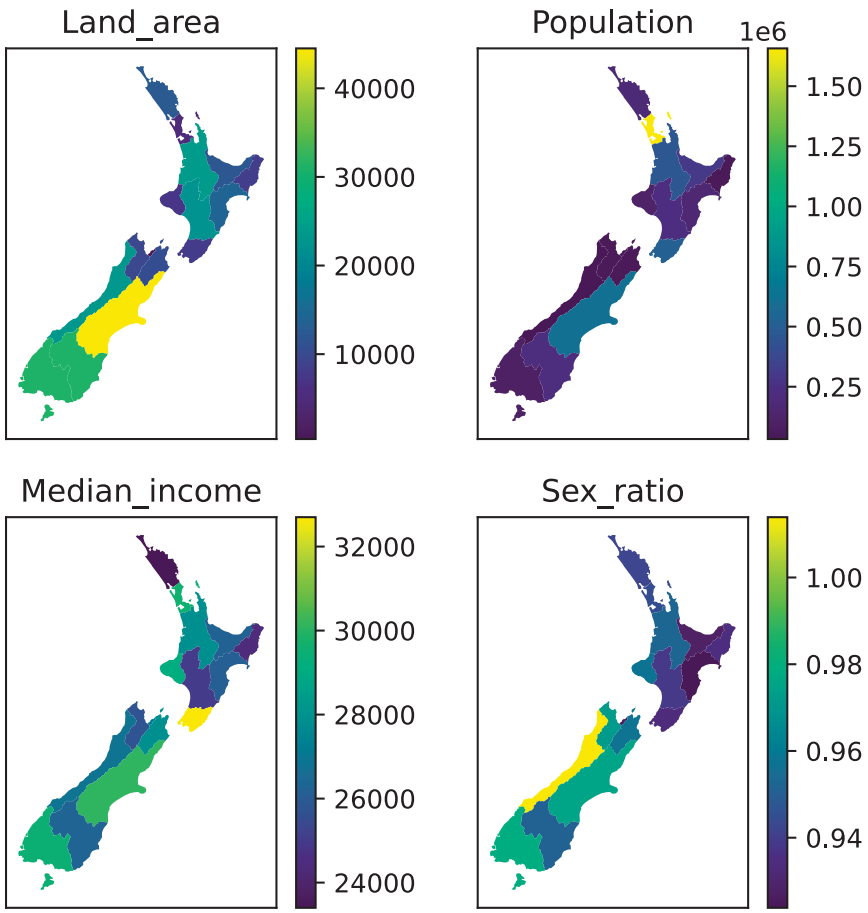


Figure 8.19: Two-dimensional layout in a faceted map, using a `for` loop

```
fig, ax = plt.subplots(nrows=int(len(vars)/2), ncols=2, figsize=(6, 6))
ax = ax.flatten()
for i in range(len(vars)):
    nz.plot(ax=ax[i], column=vars[i], legend=True)
    ax[i].set_title(vars[i])
    ax[i].xaxis.set_visible(False)
    ax[i].yaxis.set_visible(False)
```

It is also possible to 'manually' specify the properties of each panel, and which row/column it goes in. This can be useful when the various panels have different components, or even completely different types of plots (e.g., Figure 5.4), making automation with a `for` loop less applicable. For example, here is a plot similar to Figure 8.19, but specifying each panel using a separate expression instead of using a `for` loop (Figure 8.20).

```
fig, ax = plt.subplots(ncols=2, nrows=int(len(vars)/2), figsize=(6, 6))
nz.plot(ax=ax[0][0], column=vars[0], legend=True)
ax[0][0].set_title(vars[0])
nz.plot(ax=ax[0][1], column=vars[1], legend=True)
ax[0][1].set_title(vars[1])
nz.plot(ax=ax[1][0], column=vars[2], legend=True)
ax[1][0].set_title(vars[2])
nz.plot(ax=ax[1][1], column=vars[3], legend=True)
ax[1][1].set_title(vars[3]);
```

8.2.8 Exporting

Static maps can be exported to a file using the `matplotlib.pyplot.savefig` function. For example, the following code section recreates Figure 8.14, but this time the last expression saves the image to a JPG image named `plot_geopandas.jpg`.

```
base = nz.plot(color='none')
nz_height.plot(ax=base, color='red');
plt.savefig('output/plot_geopandas.jpg')
```

Figures with rasters can be exported exactly the same way. For example, the following code section (Section 8.2.5) creates an image of a raster and a vector layer, which is then exported to a file named `plot_rasterio.jpg`.

```
fig, ax = plt.subplots(figsize=(5, 5))
rasterio.plot.show(nz_elev, ax=ax)
nz.to_crs(nz_elev.crs).plot(ax=ax, facecolor='none', edgecolor='r');
plt.savefig('output/plot_rasterio.jpg')
```

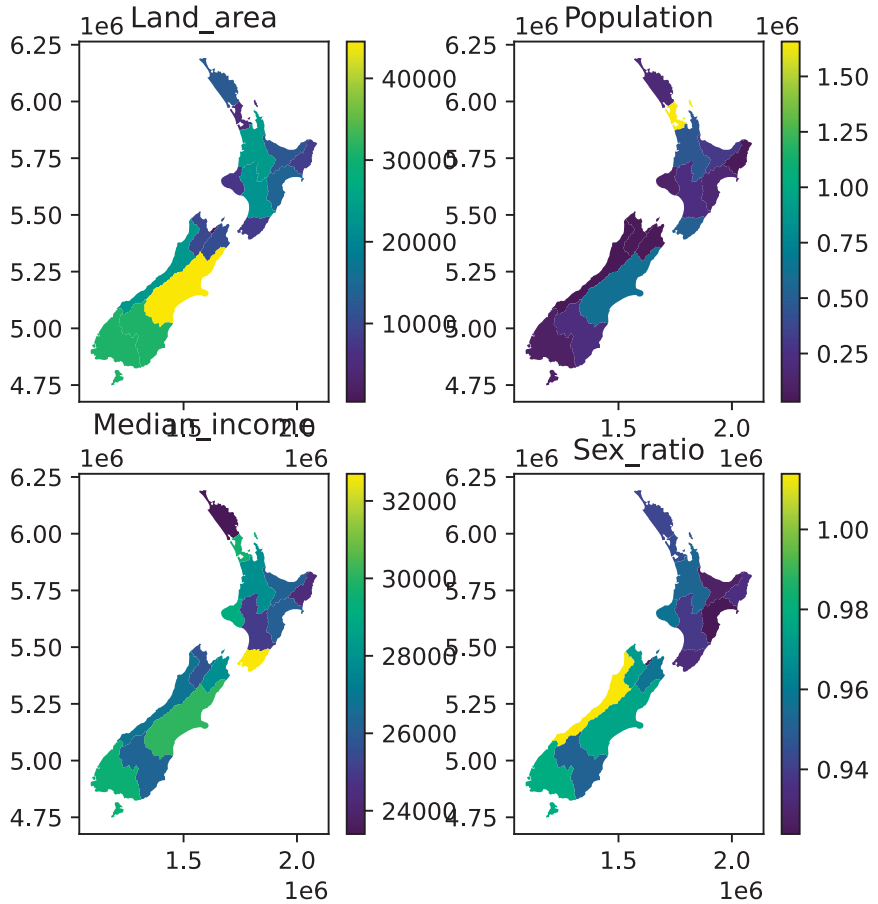


Figure 8.20: Two-dimensional layout in a faceted map, using 'manual' specification of the panels

Image file properties can be controlled through the `plt.subplots` and `plt.savefig` parameters. For example, the following code section exports the same raster plot to a file named `plot_rasterio2.svg`, which has different dimensions (width = 5 *in*, height = 7 *in*), a different format (SVG), and different resolution (300 *DPI*).

```
fig, ax = plt.subplots(figsize=(5, 7))
rasterio.plot.show(nz_elev, ax=ax)
nz.to_crs(nz_elev.crs).plot(ax=ax, facecolor='none', edgecolor='r');
plt.savefig('output/plot_rasterio2.svg', dpi=300)
```

8.3 Interactive maps

While static maps can enliven geographic datasets, interactive maps can take them to a new level. Interactivity can take many forms, the most common and useful of which is the ability to pan around and zoom into any part of a geographic dataset overlaid on a 'web map' to show context. Less advanced interactivity levels include popups which appear when you click on different features, a kind of interactive label. More advanced levels of interactivity include the ability to tilt and rotate maps, and the provision of 'dynamically linked' sub-plots which automatically update when the user pans and zooms (Pezanowski et al. 2018).

The most important type of interactivity, however, is the display of geographic data on interactive or 'slippy' web maps. Significant features of web maps are that (1) they eventually comprise static HTML files, easily shared and accessed by a wide audience, and (2) they can 'grab' content (e.g., basemaps) or use services from other locations on the internet, that way providing detailed context without requiring much effort from the person who created the map. The most popular approaches for web mapping, in Python and elsewhere, are based on the Leaflet JavaScript library (Dorman 2020). The **folium** Python package provides an extensive interface to create customized web maps based on Leaflet; it is recommended for highly customized maps. However, the **geopandas** wrapper `.explore`, introduced in Section 1.2.2, can be used for a wide range of scenarios which are often sufficient. This is what we cover in this section.

8.3.1 Minimal example

An interactive map of a `GeoSeries` or `GeoDataFrame` can be created with `.explore` (Section 1.2.2).

```
nz.explore()
```

8.3.2 Styling

The `.explore` method has a `color` parameter which affects both the fill and outline color. Other styling properties are specified using a `dict` through `style_kwds` (for general properties) and the `marker_kwds` (point-layer specific properties), as follows.

The `style_kwds` keys are mostly used to control the color and opacity of the outline and the fill:

- `stroke`—Whether to draw the outline
- `color`—Outline color

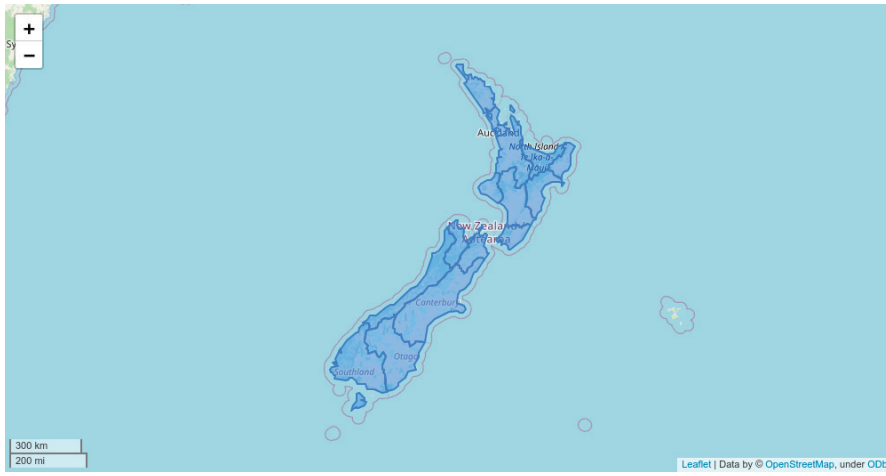


Figure 8.21: Minimal example of an interactive vector layer plot with `.explore`

- `weight`—Outline width (in pixels)
- `opacity`—Outline opacity (from `0` to `1`)
- `fill`—Whether to draw fill
- `fillColor`—Fill color
- `fillOpacity`—Fill opacity (from `0` to `1`)

For example, here is how we can set green fill color and 30% opaque black outline of `nz` polygons in `.explore` (Figure 8.22).

```
nz.explore(color='green', style_kwds={'color':'black', 'opacity':0.3})
```

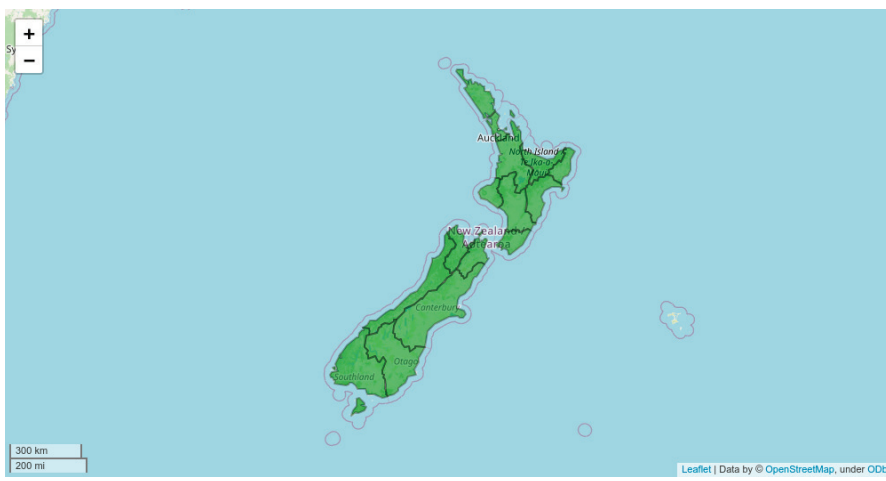


Figure 8.22: Styling of polygons in `.explore`

The `dict` passed to `marker_kwds` controls the way that points are displayed:

- `radius`—Curcle radius, in m for `circle` (see below), or in pixels for `circle_marker`
- `fill`—Whether to draw fill (for `circle` or `circle_marker`)

Accordingly, for points, we can set the `marker_type`, to one of:

- `'marker'`—A PNG image of a marker
- `'circle'`—A vector circle with radius specified in m
- `'circle_marker'`—A vector circle with radius specified in pixels (the default)

For example, the following expression draws `'circle_marker'` points with 20-pixel radius, green fill, and black outline (Figure 8.23).

```
nz_height.explore(
    color='green',
    style_kwds={'color':'black', 'opacity':0.5, 'fillOpacity':0.1},
    marker_kwds={'radius':20}
)
```

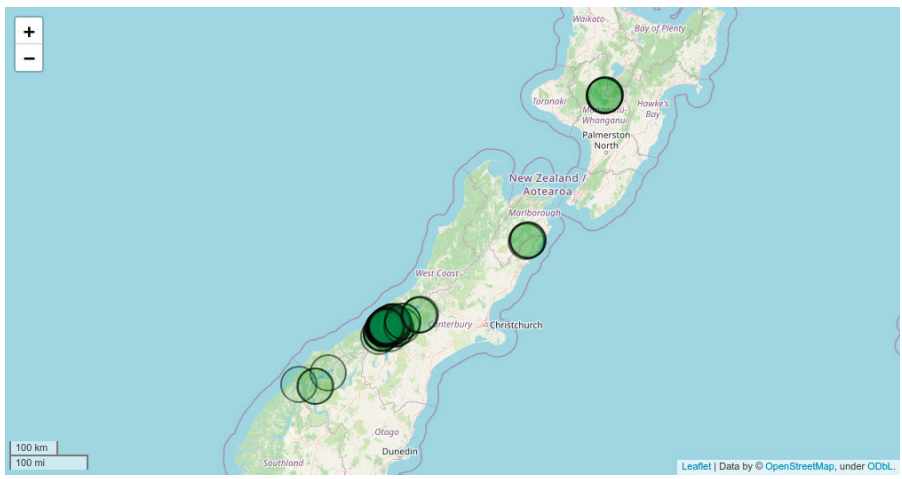


Figure 8.23: Styling of points in `.explore` (using `'circle_marker'`)

Figure 8.24 demonstrates the `'marker'` option. Note that the above-mentioned styling properties (other than `opacity`) are not applicable when using `marker_type='marker'`, because the markers are fixed PNG images.

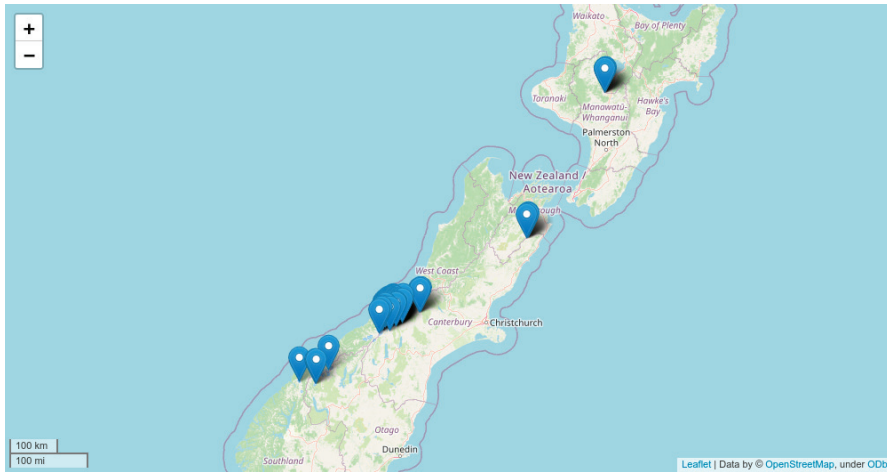


Figure 8.24: Styling of points in `.explore` (using `'marker'`)

```
nz_height.explore(marker_type='marker')
```

8.3.3 Layers

To display multiple layers, one on top of another, with `.explore`, we use the `m` argument, which stands for the previous map (Figure 8.25).

```
m = nz.explore()
nz_height.explore(m=m, color='red')
```

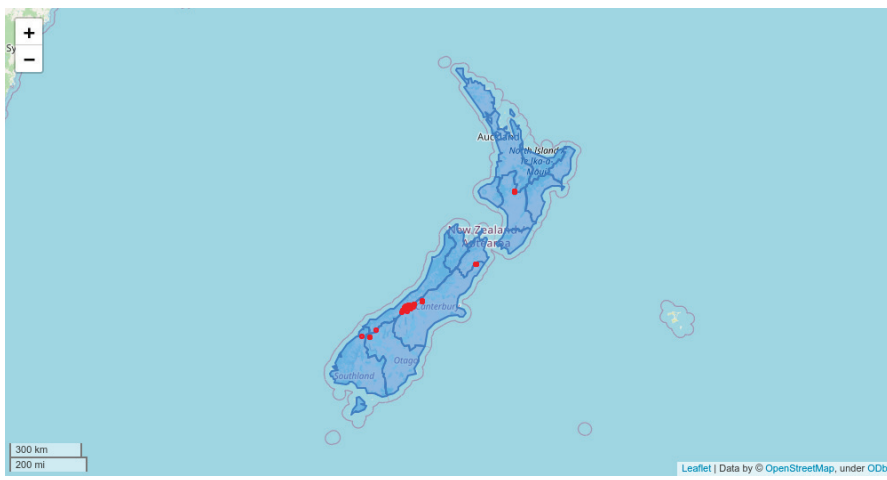


Figure 8.25: Displaying multiple layers in an interactive map with `.explore`

One of the advantages of interactive maps is the ability to turn layers 'on' and 'off'. This capability is implemented in `folium.LayerControl` from package **folium**, which the **geopandas** `.explore` method is a wrapper of. For example, this is how we can add a layer control for the `nz` and `nz_height` layers (Figure 8.26). Note the `name` properties, used to specify layer names in the control, and the `collapsed` property, used to specify whether the control is fully visible at all times (`False`), or only on mouse hover (`True`, the default).

```
m = nz.explore(name='Polygons (adm. areas)')
nz_height.explore(m=m, color='red', name='Points (elevation)')
folium.LayerControl(collapsed=False).add_to(m)
m
```

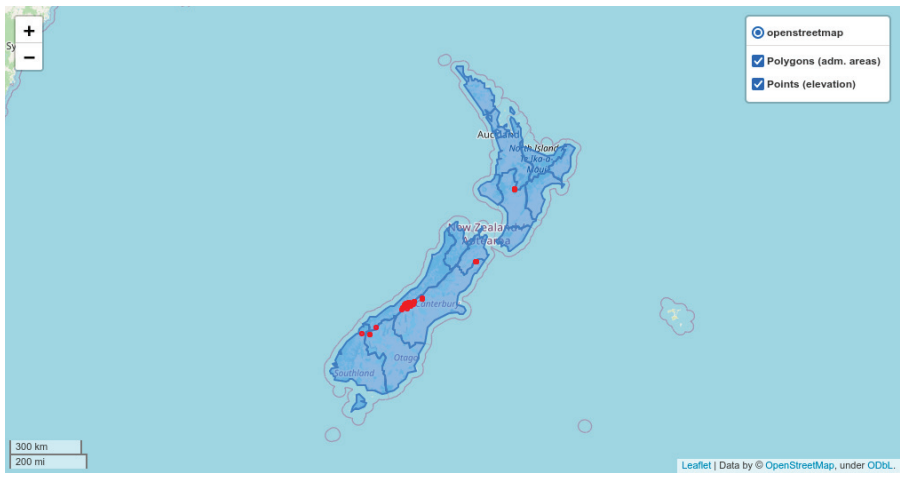


Figure 8.26: Displaying multiple layers in an interactive map with `.explore`, with layer controls

8.3.4 Symbology

Symbology can be specified in `.explore` using similar arguments as in `.plot` (Section 8.2.3). For example, Figure 8.27 is an interactive version of Figure 8.6 (a).

```
nz.explore(column='Median_income', legend=True, cmap='Reds')
```

Fixed styling (Section 8.3.4) can be combined with symbology settings. For example, polygon outline colors in Figure 8.27 are styled according to `'Median_income'`, however, this layer has overlapping outlines and their color is arbitrarily set according to the order of features (top-most features), which may be misleading and confusing. To specify fixed outline colors (e.g., black), we can use the `color` and `weight` properties of `style_kwds` (Figure 8.28):

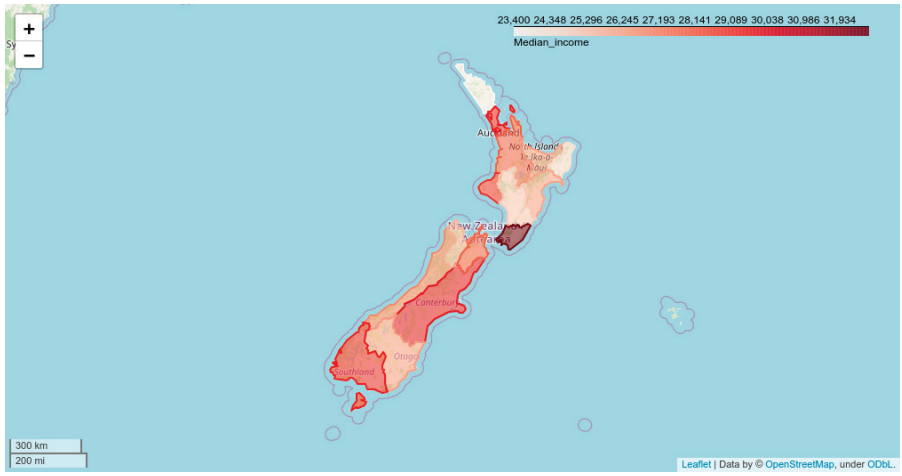


Figure 8.27: Symbology in an interactive map of a vector layer, created with `.explore`

```
nz.explore(
    column='Median_income',
    legend=True,
    cmap='Reds',
    style_kwds={'color':'black', 'weight': 0.5}
)
```

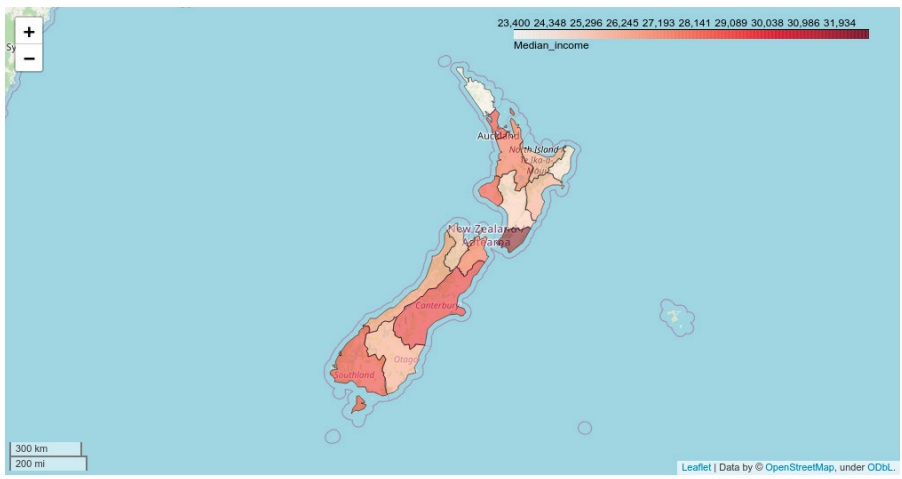


Figure 8.28: Symbology combined with fixed styling in `.explore`

8.3.5 Basemaps

The basemap in `.explore` can be specified using the `tiles` argument. Several popular built-in basemaps can be specified using a string:

- `'OpenStreetMap'`
- `'CartoDB positron'`
- `'CartoDB dark_matter'`

Other basemaps are available through the **xyzservices** package (see `xyzservices.providers` for a list), or using a custom tile server URL. For example, the following expression displays the `'CartoDB positron'` tiles in an `.explore` map (Figure 8.29).

```
nz.explore(tiles='CartoDB positron')
```

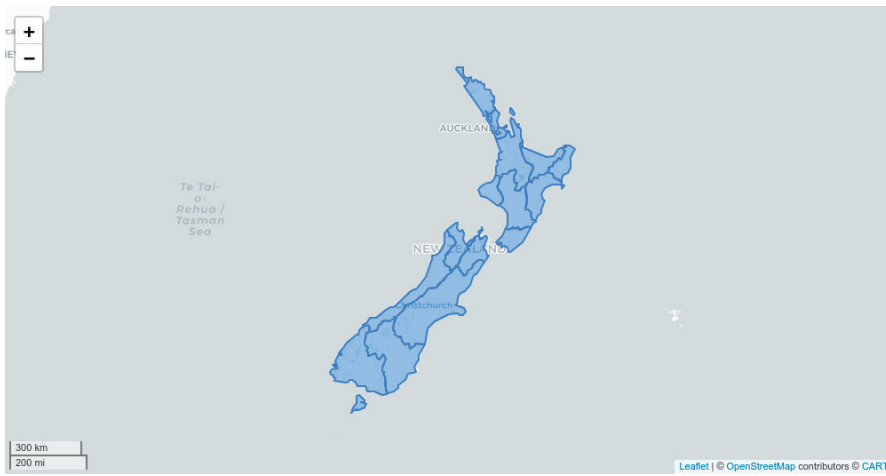


Figure 8.29: Specifying the basemap in `.explore`

8.3.6 Exporting

An interactive map can be exported to an HTML file using the `.save` method of the `map` object. The HTML file can then be shared with other people, or published on a server and shared through a URL[6]. A good free option for publishing a web map is through GitHub Pages.

[6]The GeoJSON representation of the data is embedded in the HTML file, which means that the file size can get large, and the web map may become unusable due to browser performance limitations.

For example, here is how we can export the map shown in Figure 8.26, to a file named `map.html`.

```
m = nz.explore(name='Polygons (adm. areas)')
nz_height.explore(m=m, color='red', name='Points (elevation)')
folium.LayerControl(collapsed=False).add_to(m)
m.save('output/map.html')
```

References

Bivand, Roger. 2021. "Progress in the R Ecosystem for Representing and Handling Spatial Data." *Journal of Geographical Systems* 23 (4): 515–46. https://doi.org/ghnwg3.

Bivand, Roger, Edzer Pebesma, and Virgilio Gómez-Rubio. 2013. *Applied Spatial Data Analysis with R*. Vol. 747248717. Springer. https://books.google.com?id=v0eIU9ObJXgC.

Boeing, Geoff. 2017. "OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks." *Computers, Environment and Urban Systems* 65: 126–39. https://doi.org/https://doi.org/10.1016/j.compenvurbsys.2017.05.004.

Bossche, Joris Van den, Kelsey Jordahl, Martin Fleischmann, James McBride, Jacob Wasserman, Matt Richards, Adrian Garcia Badaracco, et al. 2023. "Geopandas/Geopandas: V0.14.0." Zenodo. https://doi.org/10.5281/zenodo.8348034.

Brewer, Cynthia A. 2015. *Designing Better Maps: A Guide for GIS Users*. Second. Redlands, California: Esri Press. http://esripress.esri.com/storage/esripress/images/293/betmaped2_chapter%201.pdf.

Burrough, P. A., Rachael McDonnell, and Christopher D. Lloyd. 2015. *Principles of Geographical Information Systems*. Third. Oxford, New York: Oxford University Press.

Coppock, J Terry, and David W Rhind. 1991. "The History of GIS." *Geographical Information Systems: Principles and Applications, Vol. 1*. 1 (1): 21–43. https://www.geos.ed.ac.uk/~gisteac/ilw/generic_resources/books_and_papers/Thx1ARTICLE.pdf.

Dieck, Tammo tom. 2008. *Algebraic Topology*. EMS Textbooks in Mathematics. Zürich: European Mathematical Society. https://www.maths.ed.ac.uk/~v1ranick/papers/diecktop.pdf.

Dorman, Michael. 2020. *Introduction to Web Mapping*. CRC Press. https://geobgu.xyz/web-mapping/.

Douglas, David H, and Thomas K Peucker. 1973. "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or Its Caricature." *Cartographica: The International Journal for Geographic Information and Geovisualization* 10 (2): 112–22. https://doi.org/bjwv52.

Egenhofer, Max, and John Herring. 1990. "A Mathematical Framework for the Definition of Topological Relations." In *Proc. The Fourth International Symposium on Spatial Data Handing*, 803–13.

Gillies, Sean et al. 2007--. "Shapely: Manipulation and Analysis of Geometric Objects." toblerity.org. https://github.com/Toblerity/Shapely.

Grolemund, Garrett, and Hadley Wickham. 2016. *R for Data Science*. O'Reilly Media.

Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, et al. 2020. "Array Programming with NumPy." *Nature* 585 (7825): 357–62. https://doi.org/10.1038/s41586-020-2649-2.

Hengl, T. 2021. "Global MODIS-based Snow Cover Monthly Long-Term (2000-2012) at 500 m, and Aggregated Monthly Values (2000-2020) at 1 Km." Zenodo. https://doi.org/10.5281/zenodo.5774954.

Horn, B. K. P. 1981. "Hill Shading and the Reflectance Map." *Proceedings of the IEEE* 69 (1): 14–47. https://doi.org/10.1109/PROC.1981.11918.

Hunter, J. D. 2007. "Matplotlib: A 2D Graphics Environment." *Computing in Science & Engineering* 9 (3): 90–95. https://doi.org/10.1109/MCSE.2007.55.

Ince, E. S., F. Barthelmes, S. Reißland, K. Elger, C. Förste, F. Flechtner, and H. Schuh. 2019. "ICGEM – 15 Years of Successful Collection and Distribution of Global Gravitational Models, Associated Services, and Future Plans." *Earth System Science Data* 11 (2): 647–74. https://doi.org/gg5tzm.

Jenny, Bernhard, Bojan Šavrič, Nicholas D Arnold, Brooke E Marston, and Charles A Preppernau. 2017. "A Guide to Selecting Map Projections for World and Hemisphere Maps." In *Choosing a Map Projection*, edited by Miljenko Lapaine and Lynn Usery, 213–28. Springer.

Liu, Jian-Guo, and Philippa J. Mason. 2009. *Essential Image Processing and GIS for Remote Sensing*. Chichester, West Sussex, UK, Hoboken, NJ: Wiley-Blackwell.

Lovelace, Robin, Jakub Nowosad, and Jannes Muenchow. 2019. *Geocomputation with R*. CRC Press. https://r.geocompx.org.

Maling, D. H. 1992. *Coordinate Systems and Map Projections*. Second. Oxford, New York: Pergamon Press.

McKinney, Wes. 2010. "Data Structures for Statistical Computing in Python." In *Proceedings of the 9th Python in Science Conference*, edited by Stéfan van der Walt and Jarrod Millman, 56–61. https://doi.org/10.25080/Majora-92bf1922-00a.

Met Office. 2010-2015. *Cartopy: A Cartographic Python Library with a Matplotlib Interface*. Exeter, Devon. https://scitools.org.uk/cartopy.

Nolan, Deborah, and Duncan Temple Lang. 2014. *XML and Web Technologies for Data Sciences with R*. Use R! New York, NY: Springer.

Open Geospatial Consortium. 2019. "Well-Known Text Representation of Coordinate Reference Systems." Implementation Standard 18-010r7. Geographic Information. Open Geospatial Consortium. https://docs.opengeospatial.org/is/18-010r7/18-010r7.html.

Pebesma, Edzer, and Roger Bivand. 2022. *Spatial Data Science with Applications in R*. https://r-spatial.org/book.

———. 2023. *Spatial Data Science: With Applications in R*. CRC Press. https://r-spatial.org/book/.

Pezanowski, Scott, Alan M MacEachren, Alexander Savelyev, and Anthony C Robinson. 2018. "SensePlace3: A Geovisual Framework to Analyze Place–Time–Attribute Information in Social Media." *Cartography and Geographic Information Science* 45 (5): 420–37. https://doi.org/gc95n9.

Qiu, Fang, Caiyun Zhang, and Yuhong Zhou. 2012. "The Development of an Areal Interpolation ArcGIS Extension and a Comparative Study." *GIScience & Remote Sensing* 49 (5): 644–63. https://doi.org/gkb5fn.

Šavrič, Bojan, Bernhard Jenny, and Helen Jenny. 2016. "Projection Wizard – An Online Map Projection Selection Tool." *The Cartographic Journal* 53 (2): 177–85. https://doi.org/ggsx6z.

Spanier, Edwin Henry. 1995. *Algebraic Topology*. 1. corr. Springer ed. New York Berlin Barcelona Budapest: Springer.

Talbert, Richard J. A. 2014. *Ancient Perspectives: Maps and Their Place in Mesopotamia, Egypt, Greece, and Rome*. University of Chicago Press. https://books.google.com?id=srTbAgAAQBAJ.

Tennekes, Martijn, and Jakub Nowosad. 2022. *Elegant and Informative Maps with Tmap*. https://r-tmap.github.io/tmap-book/.

Tobler, Waldo R. 1979. "Smooth Pycnophylactic Interpolation for Geographical Regions." *Journal of the American Statistical Association* 74 (367): 519–30. https://doi.org/ghz78f.

Tomlin, C. Dana. 1990. *Geographic Information Systems and Cartographic Modeling*. Englewood Cliffs, N.J: Prentice Hall.

———. 1994. "Map Algebra: One Perspective." *Landscape and Urban Planning* 30 (1-2): 3–12. https://doi.org/dm2qm2.

Virtanen, Pauli, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, et al. 2020. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." *Nature Methods* 17: 261–72. https://doi.org/10.1038/s41592-019-0686-2.

Zevenbergen, Lyle W., and Colin R. Thorne. 1987. "Quantitative Analysis of Land Surface Topography." *Earth Surface Processes and Landforms* 12 (1): 47–56. https://doi.org/https://doi.org/10.1002/esp.3290120107.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Index

Note: **Bold** page numbers refer to **tables** and *Italic* page numbers refer to *figures.*