



4TH EDITION

# Learning Geospatial Analysis with Python

Unleash the power of Python 3 with practical techniques for learning GIS and remote sensing



JOEL LAWHEAD

# **Learning Geospatial Analysis with Python**

Unleash the power of Python 3 with practical techniques for learning GIS and remote sensing

**Joel Lawhead**



BIRMINGHAM—MUMBAI

# Learning Geospatial Analysis with Python

Copyright © 2023 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Group Product Manager:** Kaustubh Manglurkar

**Publishing Product Manager:** Apeksha Shetty

**Book Project Manager:** Farheen Fathima

**Senior Editor:** Rohit Singh

**Technical Editor:** Devanshi Ayare

**Copy Editor:** Safis Editing

**Proofreader:** Safis Editing

**Indexer:** Pratik Shirodkar

**Production Designer:** Joshua Misquitta

**DevRel Marketing Executive:** Nivedita Singh

First published: October 2013

Second edition: December 2015

Third edition: September 2019

Fourth edition: November 2023

Production reference: 1201023

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83763-917-5

[www.packtpub.com](http://www.packtpub.com)

*On the 10<sup>th</sup> anniversary of the first edition of this book, I'd like to thank everyone who has helped me along the way to be successful, including my family, friends, and coworkers. Each edition is a long process that takes the better part of a year, requiring most of my free time and focus. With each update over the years, everyone involved in this process or affected by my distraction has been very supportive, and I am truly grateful to them, especially my wife, Julie, my daughters, Lauren, Lillie, and Lainie, and my only son, Will.*

*- Joel Lawhead*

# Contributors

## About the author

**Joel Lawhead** is a PMI-certified **Project Management Professional (PMP)**, a certified **GIS Professional (GISP)**, and vice president of NVision Solutions, Inc., an award-winning firm specializing in geospatial technology integration and sensor engineering for NASA, FEMA, NOAA, the US Navy, and many other commercial and non-profit organizations. Joel began using Python in 1997 and started combining it with geospatial software development in 2000. He has authored multiple editions of *Learning Geospatial Analysis with Python* and *QGIS Python Programming Cookbook*, both from Packt. He is also the developer of the open source **Python Shapefile Library (PyShp)** and maintains a geospatial technical blog at [GeospatialPython.com](http://GeospatialPython.com).

# Acknowledgments

Writing a book is never a solitary endeavor; it's a collaborative process that involves a multitude of talented individuals working diligently behind the scenes. As we unveil the fourth edition of this work, it's only fitting to extend my heartfelt gratitude to the editorial and production team at Packt Publishing for their unwavering support and expertise. This is my fifth book for Packt Publishing, and the experience of writing technical books has altered the path of my life and career.

First and foremost, I'd like to express my deepest appreciation to Rohit Singh, our Senior Editor. Rohit, your patience and innovative suggestions have been invaluable. Your attention to detail has not only improved the quality of this book but also helped make the fourth edition the best one yet. Your leadership has been a guiding light throughout this journey.

Equally deserving of special mention are Farheen Fathima, our Book Project Manager, and Apeksha Shetty, our Publishing Product Manager. Farheen, your patience and organizational skills have kept this project on track. Apeksha, your innovative ideas have added a fresh perspective to this decade-long series.

The rest of the team has been instrumental in refining this book to its current state, including Reshma Raman (Group Product Manager), Devanshi Ayare (Technical Editor), Safis Editing (Copy Editor and Proofreader), Pratik Shirodkar (Indexer) Joshua Misquitta (Production Designer) and Nivedita Singh (Developer Relations Marketing Executive).

Thank you to the technical reviewers, Emmanuel Jolaiya and Kate Mai. Updating a book for the fourth time can be a messy process, and they did an amazing job turning it into an excellent book, thanks to their deep subject-matter expertise.

To the entire team at Packt Publishing, thank you for your collective efforts, your professionalism, and your commitment to excellence. This book would not be what it is without each and every one of you. It is only fitting that a successful book about geography owes its success to an international team from across the globe.

## About the reviewers

**Emmanuel Jolaiya** is an experienced Earth observation and geospatial software engineer. He is currently pursuing a master's degree in geospatial technologies at Universitat Jaume I, Spain, on an Erasmus Mundus scholarship. He has consulted for several top organizations, including the World Bank and Integration Environment and Energy, where he uses geospatial technology to support electrification planning in Nigeria.

Jolaiya is the founder of Spatialnode, a platform where geospatial professionals can showcase their work and connect with opportunities. His interests include geospatial community development, cloud-native geospatial, spatial SQL, and GeoAI. He has also reviewed the book *Geospatial Analysis with SQL* by Bonny P McClain.

**Kate Mai** is a GIS solutions architect who has been in the IT industry for over 10 years. She has designed and implemented GIS solutions for various organizations, including government agencies and environmental research institutes. Her work has involved harnessing geospatial data to optimize decision-making processes, improve workflows, and enhance web application products. Her expertise includes web map application development, GIS integration, and system optimization. She is also interested in what the next big thing may be in the IoT, AI, and GIS fields.

*To Lily, my puppy – thanks for your all time, support, and snuggles.*

# Table of Contents

## Preface

xv

## Part 1: The History and the Present of the Industry

1

Learning about Geospatial Analysis with Python		3	
Technical requirements	4	Map projections	25
Geospatial analysis and our world	4	Rendering	26
History of geospatial analysis	6	Remote sensing concepts	27
Evolution of Geographic Information Systems (GISs)	10	Images as data	27
		Remote sensing and color	28
Remote sensing	11	Common vector GIS concepts	28
Point cloud data	16	Data structures	28
Computer-aided drafting	19	Buffer	31
Geospatial analysis and computer programming	19	Dissolve	31
Object-oriented programming for geospatial analysis	20	Generalize	32
		Intersection	32
		Merge	33
The importance of geospatial analysis	21	Point in polygon	34
		Union	34
GIS concepts	22	Join	35
Thematic maps	22	Common raster data concepts	35
Spatial databases	24	Band math	36
Spatial indexing	24	Change detection	36
Metadata	24	Histogram	37

Feature extraction	37	Getting started with Python	38
Supervised and unsupervised classification	38	Building a SimpleGIS	39
<b>Creating the simplest possible Python GIS</b>	<b>38</b>	<b>Summary</b>	<b>47</b>
		<b>Questions</b>	<b>47</b>
		<b>Further reading</b>	<b>47</b>

## 2

<b>Learning about Geospatial Data</b>	<b>49</b>
---------------------------------------	-----------

Technical requirements	50	TIFF files	68
Overview of common data formats	50	JPEG, GIF, BMP, and PNG	69
Understanding data structures	53	Compressed formats	69
Common traits	54	ASCII grids	69
Understanding spatial indexing	54	World files	70
Spatial indexing algorithms	54	<b>What is point cloud data?</b>	<b>73</b>
What are overviews?	57	LIDAR	73
What is metadata?	58	<b>More realistic geospatial models with 3D data</b>	<b>74</b>
Understanding the file structure	58	What are web services?	75
Knowing about the most widely used vector data types	60	Understanding geospatial databases	76
Shapefiles	61	Sharing data with interchange formats	76
CAD files	63	Introducing spatiotemporal data	77
Tag-based and markup-based formats	64	Summary	79
GeoJSON	66	Questions	79
GeoPackage	67	Further reading	79
Understanding raster data types	68		

## 3

<b>The Geospatial Technology Landscape</b>	<b>81</b>
--	-----------

Technical requirements	82	PDAL	87
Understanding data access	82	<b>Understanding computational geometry</b>	
GDAL	84		87

---

The PROJ projection library	88	NASA WorldWind	108
CGAL	89	ArcGIS	110
JTS	90	Leaflet and OpenLayers	111
GEOS	91	<b>Understanding metadata management</b>	<b>112</b>
PostGIS	92	Python's pycsw library	112
Other spatially enabled databases	96	GeoNode	112
Routing	101	GeoNetwork	113
<b>Understanding desktop tools (including visualization)</b>	<b>103</b>	<b>A quick look at artificial intelligence</b>	<b>114</b>
Quantum GIS	103	<b>Summary</b>	<b>115</b>
GRASS GIS	104	<b>Questions</b>	<b>116</b>
gvSIG	105	<b>Further reading</b>	<b>116</b>
OpenJUMP	106		
Google Earth	106		

## Part 2: Geospatial Analysis Concepts

### 4

---

<b>Geospatial Python Toolbox</b>		<b>119</b>	
Technical requirements	120	The minidom module	134
Using QGIS	121	The ElementTree module	136
Installing third-party Python modules	122	Building XML using ElementTree and minidom	137
Anaconda	122	<b>Well-Known Text (WKT)</b>	<b>141</b>
<b>Jupyter</b>	<b>123</b>	<b>Python JSON libraries</b>	<b>143</b>
PyPI and pip	124	The json module	144
The Python virtualenv module	124	The geojson module	146
<b>Python networking libraries for acquiring data</b>		<b>OGR</b>	<b>147</b>
The Python urllib module	125	<b>PyShp</b>	<b>148</b>
The Python requests module	126	<b>Shapely</b>	<b>149</b>
FTP	128	<b>Fiona</b>	<b>149</b>
<b>Bundling and compressing files</b>	<b>129</b>	<b>GDAL</b>	<b>151</b>
<b>Python markup and tag-based parsers</b>	<b>131</b>	<b>NumPy</b>	<b>151</b>
		<b>PIL</b>	<b>153</b>
	133		

PNGCanvas	154	OSMnx	161
GeoPandas	156	Folium	161
PyFPDF	157	Summary	162
PyMySQL	158	Questions	163
Rasterio	160	Further reading	163

## 5

<b>Python and Geospatial Algorithms</b>	<b>165</b>
---	------------

---

Technical requirements	166	Understanding coordinate format conversion	181
Measuring distance	166	Calculating the area of a polygon	182
Using the Pythagorean theorem to measure distance	169	Using ChatGPT to measure a polygon perimeter	183
Using the haversine formula	171	Summary	187
Using the Vincenty formula	173	Questions	187
Calculating line direction	176	Further reading	187
Understanding coordinate conversion	176		
Understanding reprojection	178		

## 6

<b>Creating and Editing GIS Data</b>	<b>189</b>
--------------------------------------	------------

---

Technical requirements	190	Connecting polygon faces to the nearest line point	212
Editing shapefiles	190	<b>Creating images for visualization</b>	<b>216</b>
Accessing the shapefile	191	Dot density calculations	217
Changing a shapefile	198	Choropleth maps	220
Adding fields	201	Using spreadsheets	223
Merging shapefiles	201	Creating heat maps	225
Splitting shapefiles	203	<b>Using GPS data</b>	<b>226</b>
Performing selections	204		
Aggregating geometry	209		
Extracting geometry	211		

---

Turning addresses into points with geocoding	227	Summary	231
Performing GIS analysis faster with multiprocessing	229	Questions	231
		Further reading	231

**7**


---

<b>Python and Remote Sensing</b>	<b>233</b>
----------------------------------	------------

Technical requirements	234	Extracting features from images	253
Examining raster data properties	234	Understanding change detection	259
Swapping image bands	236	Extracting image footprints using ChatGPT	264
Creating image histograms	239	Summary	267
Performing a histogram stretch	243	Questions	268
Clipping images	245	Further reading	268
Classifying images	250		

**8**


---

<b>Python and Elevation Data</b>	<b>269</b>
----------------------------------	------------

Technical requirements	269	Using PIL to visualize LiDAR data	290
Accessing ASCII Grid files	270	Creating a triangulated irregular network	295
Reading grids	270	Colorizing LiDAR with aerial images	298
Writing grids	272	Classifying LiDAR	302
Creating a shaded relief	274	Working with bathymetry	304
Creating elevation contours	279	Summary	308
Working with LiDAR data	283	Questions	308
Creating a grid from the LiDAR data	284	Further reading	308

## Part 3: Practical Geospatial Processing Techniques

9

### Advanced Geospatial Modeling 311

---

Technical requirements	312	Performing least cost path analysis	333
Creating a normalized difference vegetation index (NDVI)	312	The real-world example	336
Setting up the framework	314	<b>Converting the route to a shapefile</b>	343
Loading the data	315	Routing along streets	344
Rasterizing the shapefile	316	Geolocating photos	348
Clipping the bands	317	Calculating satellite image cloud cover	351
Using the NDVI formula	318	Summary	355
Classifying the NDVI	319	Questions	355
<b>Creating a flood inundation model</b>	323	Further reading	355
The flood fill function	325		
Creating a color hillshade	330		

10

### Working with Real-Time Data 357

---

Technical requirements	358	Storm chasing	363
Limitations of real-time data	358	Gathering reports from the field	368
Using real-time data	358	Summary	371
Tracking vehicles	359	Questions	372
Getting a vehicle location	359	Further reading	372
Mapping a vehicle location	361		

**11**

<b>Putting It All Together</b>	<b>373</b>
Technical requirements	373
Understanding a typical GPS report	374
Building a GPS reporting tool	375
Importing libraries	375
Setting up logging	376
Helper functions	376
Program variables	379
Parsing the GPX file	380
Downloading the basemap and elevation data	381
Hillshading the elevation data	383
Creating a map	384
Adding a photo marker	385
Creating an elevation profile chart	386
Creating a weather report	388
Generating a PDF report	389
<b>Summary</b>	<b>391</b>
<b>Questions</b>	<b>391</b>
<b>Further reading</b>	<b>391</b>
<b>Assessments</b>	<b>393</b>
Chapter 1 – Learning about Geospatial Analysis with Python	393
Chapter 2 – Learning about Geospatial Data	393
Chapter 3 – The Geospatial Technology Landscape	393
Chapter 4 – Geospatial Python Toolbox	393
Chapter 5 – Python and Geospatial Algorithms	393
Chapter 6 – Creating and Editing GIS Data	394
Chapter 7 – Python and Remote Sensing	394
Chapter 8 – Python and Elevation Data	394
Chapter 9 – Advanced Geospatial Modeling	394
Chapter 10 – Working with Real-Time Data	394
Chapter 11 – Putting It All Together	394
<b>Index</b>	<b>395</b>
<b>Other Books You May Enjoy</b>	<b>408</b>



# Preface

This book starts by giving you the background of geospatial analysis and then offers a flow of the techniques and technology used, splitting the field into its component specialty areas, such as **Geographic Information Systems (GISs)**, remote sensing, elevation data, advanced modeling, and real-time data. The focus of the book is to lay a strong foundation in using the powerful Python language and framework to approach geospatial analysis effectively. In doing so, we'll focus on using pure Python, as well as certain Python tools and APIs, and generic algorithms. You will be able to analyze various forms of geospatial data, learn about real-time data tracking, and see how to apply what you learn to interesting scenarios.

While many third-party geospatial libraries are used throughout the examples, a special effort will be made by us to use pure Python, with no dependencies, whenever possible. This focus on pure Python 3 examples is what will set this book apart from nearly all other resources in this field. We will also go through some popular libraries that weren't in the previous version of the book.

## Who this book is for

This book is for Python developers, researchers, or analysts who want to perform geospatial modeling and GIS analysis with Python. Basic knowledge of digital mapping and analysis using Python or other scripting languages will be helpful.

## What this book covers

*Chapter 1, Learning about Geospatial Analysis with Python*, introduces geospatial analysis as a way of answering questions about our world. The differences between GIS and remote sensing are explained. Common geospatial analysis processes are demonstrated, using illustrations, basic formulas, pseudo code, and Python.

*Chapter 2, Learning about Geospatial Data*, explains the major categories of data and several newer formats that are becoming more and more common. Geospatial data comes in many forms. The most challenging part of geospatial analysis is acquiring the data that you need and preparing it for analysis. Familiarity with these data types is essential to understanding geospatial analysis.

*Chapter 3, The Geospatial Technology Landscape*, tells you about the geospatial technology ecosystem, which consists of thousands of software libraries and packages. This vast array of choices is overwhelming for newcomers to geospatial analysis. The secret to learning geospatial analysis quickly is understanding the handful of libraries and packages that really matter. Most other software is derived from these critical packages. Understanding the hierarchy of geospatial software and how it's used allows you to quickly understand and evaluate any geospatial tool.

*Chapter 4, Geospatial Python Toolbox*, introduces software and libraries that form the basis of the book and are used throughout. Python’s role in the geospatial industry is explored – the GIS scripting language, the mash-up glue language, and the full-blown programming language. Code examples are used to teach data editing concepts, and many of the basic geospatial concepts in *Chapter 1, Learning about Geospatial Analysis with Python*, are also demonstrated in this chapter.

*Chapter 5, Python and Geospatial Algorithms*, explores the core of geospatial analysis by applying Python to GIS algorithms. You’ll learn essential techniques such as measuring distance, calculating line direction, and understanding coordinate conversion. With hands-on code samples, you’ll be well-equipped to tackle more advanced geospatial challenges by the end of the chapter.

*Chapter 6, Creating and Editing GIS Data*, focuses on creating and editing GIS data. You’ll learn how to manipulate shapefiles, create images for visualization, and explore geocoding and multiprocessing. By the end of the chapter, you’ll have a robust set of tools for GIS analysis, setting the stage for future topics such as remote sensing and elevation data.

*Chapter 7, Python and Remote Sensing*, shows you how to work with remote sensing geospatial data. Remote sensing includes some of the most complex and least-documented geospatial operations. This chapter will build a solid core for you and demystify remote sensing using Python.

*Chapter 8, Python and Elevation Data*, demonstrates the most common uses of elevation data and how to work with its unique properties. Elevation data deserves a chapter all on its own. Elevation data can be contained in almost any geospatial format but is used quite differently from other types of geospatial data.

*Chapter 9, Advanced Geospatial Modeling*, uses Python to teach you the true power of geospatial technology. Geospatial data editing and processing help us understand the world as it is. The true power of geospatial analysis is modeling. Geospatial models help us predict the future, narrow a vast field of choices down to the best options, and visualize concepts that cannot be directly observed in the natural world.

*Chapter 10, Working with Real-Time Data*, examines the modern phenomenon of geospatial analysis. A wise geospatial analyst once said, “As soon as a map is created it is obsolete.” Until recently, by the time you collected data about the Earth, processed it, and created a geospatial product, the world it represented had already changed. However, modern geospatial data shatters this notion. Datasets are available over the internet that are up to the minute, or even the second. This data fundamentally changes the way we perform geospatial analysis.

*Chapter 11, Putting It All Together*, brings together all the skills and techniques you’ve acquired throughout the book to create a GPS route analysis report, similar to reports found in popular fitness apps. You’ll work with nearly 500 lines of Python code to build this comprehensive report, which will include elements such as a route map, elevation profile, and even real-time weather conditions. By the end of the chapter, you’ll have a fully functional GPS reporting tool that can be used for various applications, such as fitness tracking or navigation.

## To get the most out of this book

This book assumes you have basic knowledge of the Python programming language. You will require the Anaconda 2.4.0 or higher with Python (3.10 or higher); a minimum hardware requirement of a 300-MHz processor, 128 MB of RAM, and 1.5 GB of available hard disk; and a Windows, Linux, or macOS X operating system.

Software/hardware covered in the book	Operating system requirements
Anaconda 2.4.0 or higher	Windows – 64-bit x86; macOS – 64-bit x86 and M1; or Linux – 64-bit x86, 64-bit aarch64 (AWS Graviton2), 64-bit Power8/Power9, or s390x (Linux on IBM Z and LinuxONE)
Python 3.10 or higher	8 GB of RAM
ECMAScript 11	6 GB of disk space or more

Follow the instructions for the latest Anaconda installation. All scripts in this book assume you will run the code from the command line within an Anaconda Python environment. If you try to use Jupyter Notebooks, you may get unpredictable results.

**If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.**

The GitHub repository contains folders for each chapter that include all code in the book, as well as color images and geospatial data files needed for the examples.

## Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Conventions used

There are a number of text conventions used throughout this book.

**Code in text:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “The `turtle` module uses the concept of a cursor, known as a `pen`.”

A block of code is set as follows:

```
{ "type": "GeometryCollection",
  "geometries": [
    { "type": "Point",
      "coordinates": [
        [-89.33, 30]
      ]
    },
  ]}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in **bold**:

```
if (sinSigma == 0):
    distance = 0 # coincident points
    break
cosSigma = sinU1*sinU2 + cosU1*cosU2*cosLam
sigma = math.atan2(sinSigma, cosSigma)
sinAlpha = cosU1 * cosU2 * sinLam / sinSigma
cosSqAlpha = 1 - sinAlpha**2
```

Any command-line input or output is written as follows:

```
pip install virtualenv
```

**Bold:** Indicates a new term, an important word, or words that you see on screen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “We’ll examine why you might want to learn a programming language as a geospatial analyst, as opposed to just using **Geographic Information System (GIS)** applications.”

**Tips or important notes**

Appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, email us at [customercare@packtpub.com](mailto:customercare@packtpub.com) and mention the book title in the subject of your message.

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata) and fill in the form.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packt.com](mailto:copyright@packt.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Share your thoughts

Once you've read *Cloud Penetration Testing for Red Teamers*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781837639175>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

# **Part 1:**

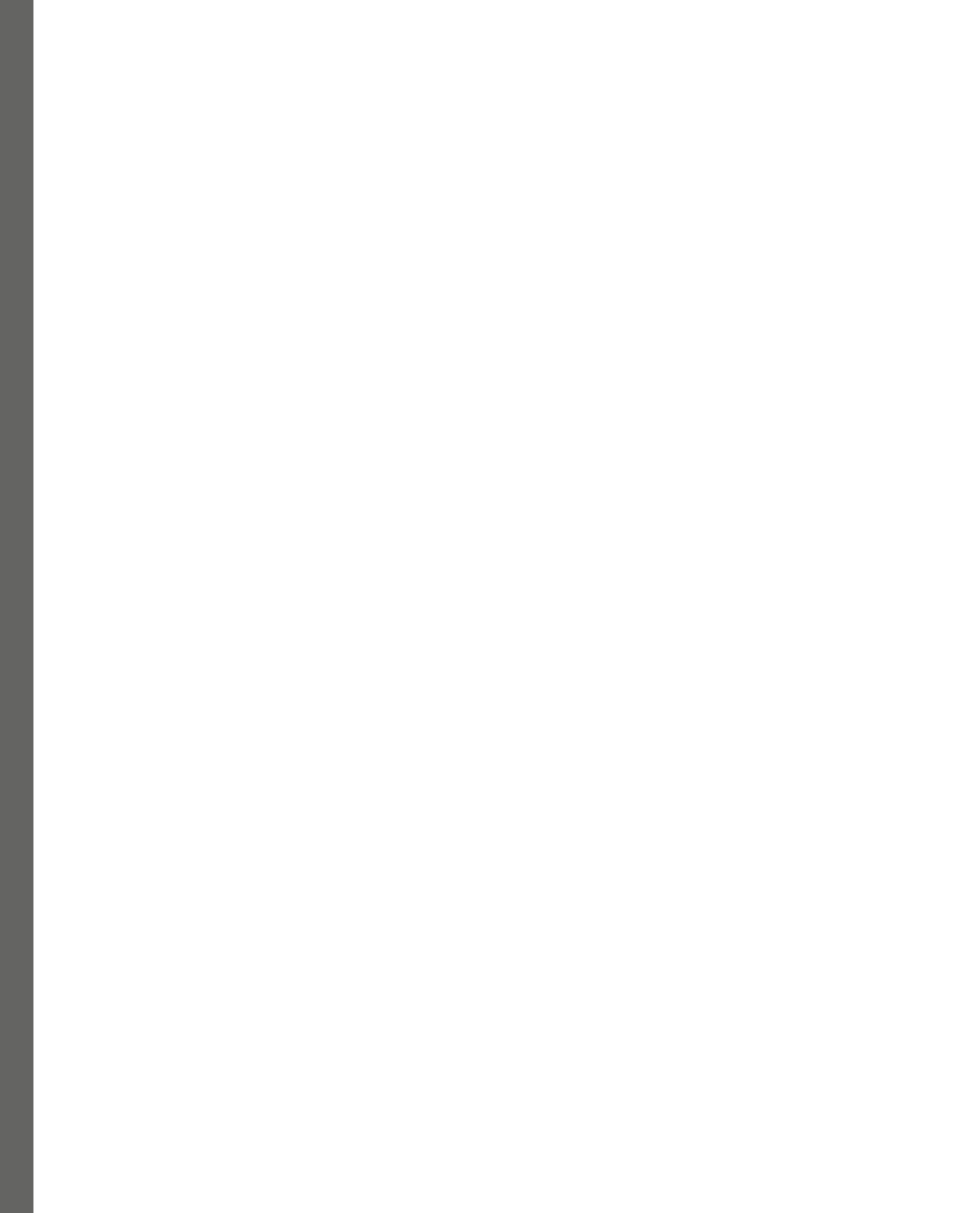
## **The History and**

## **the Present of the Industry**

This part starts by demonstrating common geospatial analysis processes using illustrations, basic formulas, simple code, and Python. Building on that, you'll learn how to play with geospatial data – acquiring data and preparing it for various analyses. After that, you'll gain an understanding of the various software packages and libraries used in the geospatial technology ecosystem. At the end of this part, you'll learn how to evaluate any geospatial tool to see whether it's what you need for your project.

This part contains the following chapters:

- *Chapter 1, Learning about Geospatial Analysis with Python*
- *Chapter 2, Learning about Geospatial Data*
- *Chapter 3, The Geospatial Technology Landscape*



# 1

# Learning about Geospatial Analysis with Python

This chapter is an overview of geospatial analysis. We will see how geospatial technology is currently impacting our world by looking at a case study of one of the worst pandemics the world has ever seen and how geospatial analysis helped track the spread of the disease to buy researchers time to create a vaccine. Next, we'll step through the history of geospatial analysis, which predates computers and even paper maps! Then, we'll examine why you might want to learn a programming language as a geospatial analyst as opposed to just using **Geographic Information System (GIS)** applications. We'll realize the importance of making geospatial analysis as accessible as possible to the broadest number of people. Then, we'll step through basic GIS and remote sensing concepts and terminology that will stay with you through the rest of this book. Finally, we'll start using Python for geospatial analysis by building the simplest possible GIS from scratch!

Here's a quick overview of the topics we'll be covering in this chapter:

- Geospatial analysis and our world
- History of geospatial analysis
- Evolution of **Geographic Information Systems (GISs)**
- Remote sensing concepts
- Point cloud data
- Computer-aided drafting
- Geospatial analysis and computer programming
- The importance of geospatial analysis
- GIS concepts
- Common GIS processes
- Common remote sensing processes

- Common raster data concepts
- Creating the simplest possible Python GIS

By the end of this chapter, you will understand geospatial analysis as a way of answering questions about our world and the differences between GIS and remote sensing.

## Technical requirements

This chapter provides a foundation for geospatial analysis, which is needed to pursue any subject in the areas of remote sensing and GIS, including the material in the rest of the chapters of this book. The code for this book can be found in the following GitHub code repository: <https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-4th-Edition>. We will be using Python 3.10.9 for the code examples, which will be provided through the Anaconda 3 platform. The code files for this chapter can be accessed on GitHub: [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730\\_01\\_Asset\\_Files](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730_01_Asset_Files).

## Geospatial analysis and our world

In December 2019, doctors reported a cluster of cases of a mysterious pneumonia-like illness in Wuhan, China. At first, it was thought to be a minor outbreak, but as the number of cases continued to rise, it quickly became clear that this was something much more serious.

As the virus began to spread to other countries, the **World Health Organization (WHO)** declared a global health emergency on January 30, 2020. Despite this warning, many countries were slow to take action, and the virus continued to spread unchecked.

By March 2020, the virus had reached pandemic proportions, with cases reported in every corner of the globe. Governments scrambled to respond, implementing lockdowns and travel bans in an attempt to slow the spread of the virus.

As the number of cases and deaths continued to rise, the world watched in horror as hospitals became overwhelmed and healthcare systems struggled to keep up. For the first time in over a century, humanity found itself in a global pandemic, battling a new virus named COVID-19.

As with any new virus, there was no vaccine or even an effective treatment. Medical experts raced to develop a vaccine. The only solution in the short term was to buy time. To do that, the world needed a way to track the virus as it spread to focus resources in the areas it raged most intensely.

In the US, at Johns Hopkins University in Baltimore, Maryland, a PhD candidate named Ensheng Dong had watched as news of the virus spread from his home country – China. As a student, Dong studied both epidemiology and a technology called GIS, a computer system that displays and analyzes geographically referenced information. Dong became worried about his family's safety, and when the first COVID case hit Washington, he wanted to take action.

The next day, he met with his faculty advisor, Dr. Lauren Gardner, who suggested he use his knowledge of epidemiology and GIS to create a dashboard that would track the virus for the world. Dong began scouring the internet for COVID data and posted it to an online map twice a day while barely sleeping in between. He posted red dots on a map with a dark background. In areas with a large number of cases, he would increase the size of the dot to show the severity of the spread. As word of the dashboard grew, Dong began receiving help to automate the data collection and posting process.

These dashboard map visualizations helped public health officials understand where the virus was spreading and identify hotspots that needed extra attention. It helped track the effectiveness of containment measures such as lockdowns and social distancing rules. It also allowed news organizations to notify the public.

The following figure shows the COVID-19 dashboard:

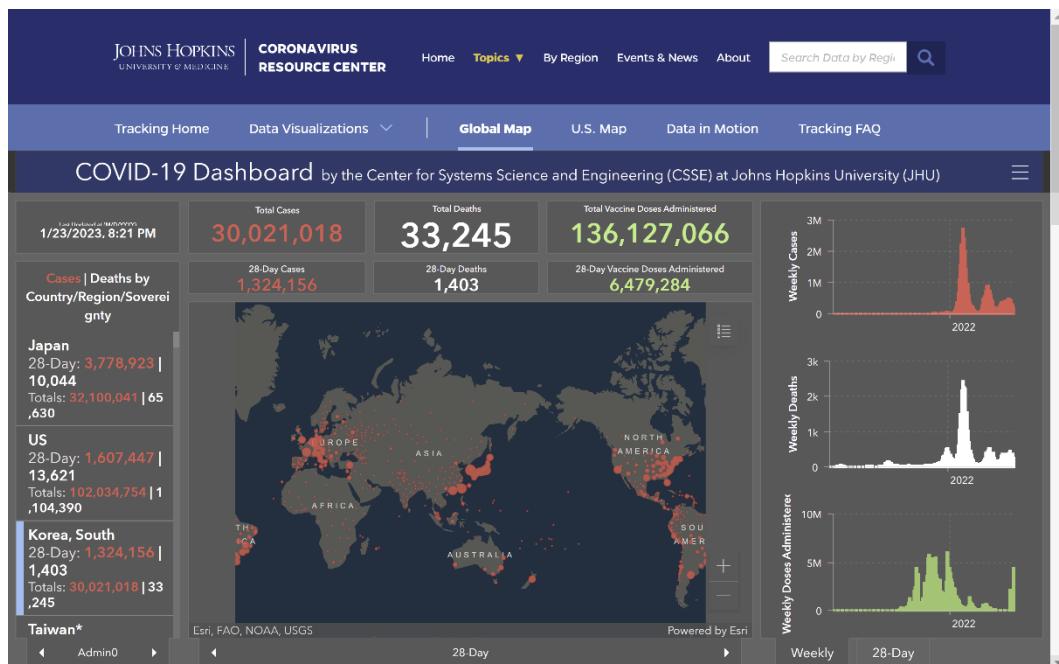


Figure 1.1 – The COVID-19 dashboard

Government organizations used other GIS maps as well in response to the pandemic to identify high-risk populations. By overlaying data on demographics, income levels, and pre-existing health conditions, GIS helped officials identify communities that were particularly vulnerable to the virus and target resources to those areas.

Officials also used GIS to help manage the logistics of the pandemic's response. For example, they used it to plan the distribution of personal protective equipment, medical supplies, and vaccines. GIS also tracked the movements of healthcare workers and other essential personnel, ensuring they were deployed to where they were needed most.

In short, GIS has played a vital role in the response to the pandemic, providing critical information and tools to help organizations respond to the crisis more effectively.

### Other uses of GIS

Geospatial analysis can be found in almost every industry, including real estate, oil and gas, agriculture, defense, politics, health, transportation, and oceanography, to name a few. For a good overview of how geospatial analysis is used in dozens of different industries, visit <https://www.esri.com/en-us/what-is-gis/overview>.

## History of geospatial analysis

Geospatial analysis can be traced back as far as 17,000 years ago, to the Lascaux cave in southwestern France. In this cave, Paleolithic artists painted commonly hunted animals and what many experts recently concluded are dots representing the animals' lunar cycles to note seasonal behavior patterns of prey, such as mating or migration. Though crude, these paintings demonstrate an ancient example of humans creating abstract models of the world around them and correlating spatial-temporal features to find relationships. The following figure shows one of these paintings – a bull with four dots on its back, cross-referencing a lunar time reference:



Figure 1.2 – A cave painting of prey tagged with a lunar cycle reference to predict when it will appear in hunting grounds again

Over the centuries, the art of cartography and the science of land surveying have developed, but it wasn't until the 1800s that significant advances in geographic analysis emerged. Deadly cholera outbreaks in Europe between 1830 and 1860 led geographers in Paris and London to use geographic analysis for epidemiological studies.

In 1854, Charles Picquet used different halftone shades of gray to represent the deaths per thousand citizens in the 48 districts of Paris as part of a report on the cholera outbreak. In 1854, Dr. John Snow expanded on this method by tracking a cholera outbreak in London as it occurred. By placing a point on a map of the city each time a fatality was diagnosed, he was able to analyze the clustering of cholera cases. Snow traced the disease to a single water pump and prevented further cases. The following zoomed-in map section has three layers with streets, a labeled dot for each pump, and bars for each cholera death in a household:



Figure 1.3 – 1854 map of London tracking a cholera outbreak, with dots for the location of water pumps that were potential sources of the disease and bars showing the number of outbreaks per household

Geospatial analysis wasn't just used for the war on diseases. For centuries, generals and historians have used maps to understand human warfare. A retired French engineer named Charles Minard produced

some of the most sophisticated infographics that were ever drawn between 1850 and 1870. The term *infographics* is too generic to describe these drawings because they have strong geographic components. The quality and detail of these maps make them fantastic examples of geographic information analysis, even by today's standards. Minard released his masterpiece in 1869:

*“La carte figurative des pertes successives en hommes de l’Armée Française dans la campagne de Russie 1812-1813,” which translates to “Figurative map of the successive losses of men of the French army in the Russian Campaign 1812-13.”*

This depicts the decimation of Napoleon's army in the Russian campaign of 1812. The map shows the size and location of the army over time, along with prevailing weather conditions. The following figure contains four different series of information on a single theme. It is a fantastic example of geographic analysis using pen and paper. The size of the army is represented by the widths of the brown and black swaths at a ratio of one millimeter for every 10,000 men. The numbers are also written along the swaths. The brown-colored path shows soldiers who entered Russia, while the black-colored path represents the ones who made it out. The map scale is shown to the right in the center as one French league (2.75 miles or 4.4 kilometers). The chart at the bottom runs from right to left and depicts the brutally freezing temperatures that were experienced by the soldiers on their march home from Russia:

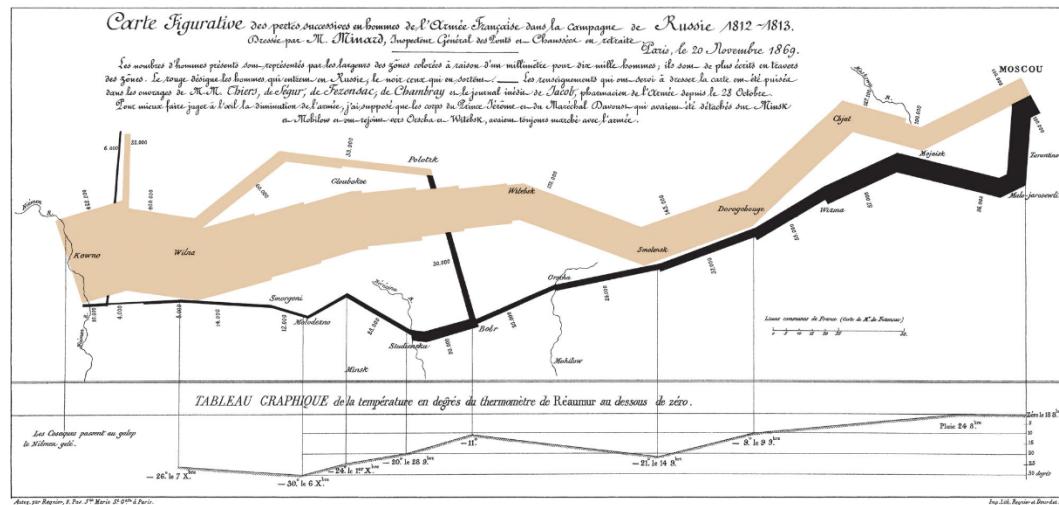


Figure 1.4 – Charles Minard's famous geographic story map showing the decimation of Napoleon's army during the Russian Campaign of 1812. It combines geography, time, and statistics

While far more mundane than a war campaign, Minard released another compelling map cataloging the number of cattle sent to Paris from around France. Minard used pie charts of varying sizes in the regions of France to show each area's variety and the volume of cattle that was shipped:

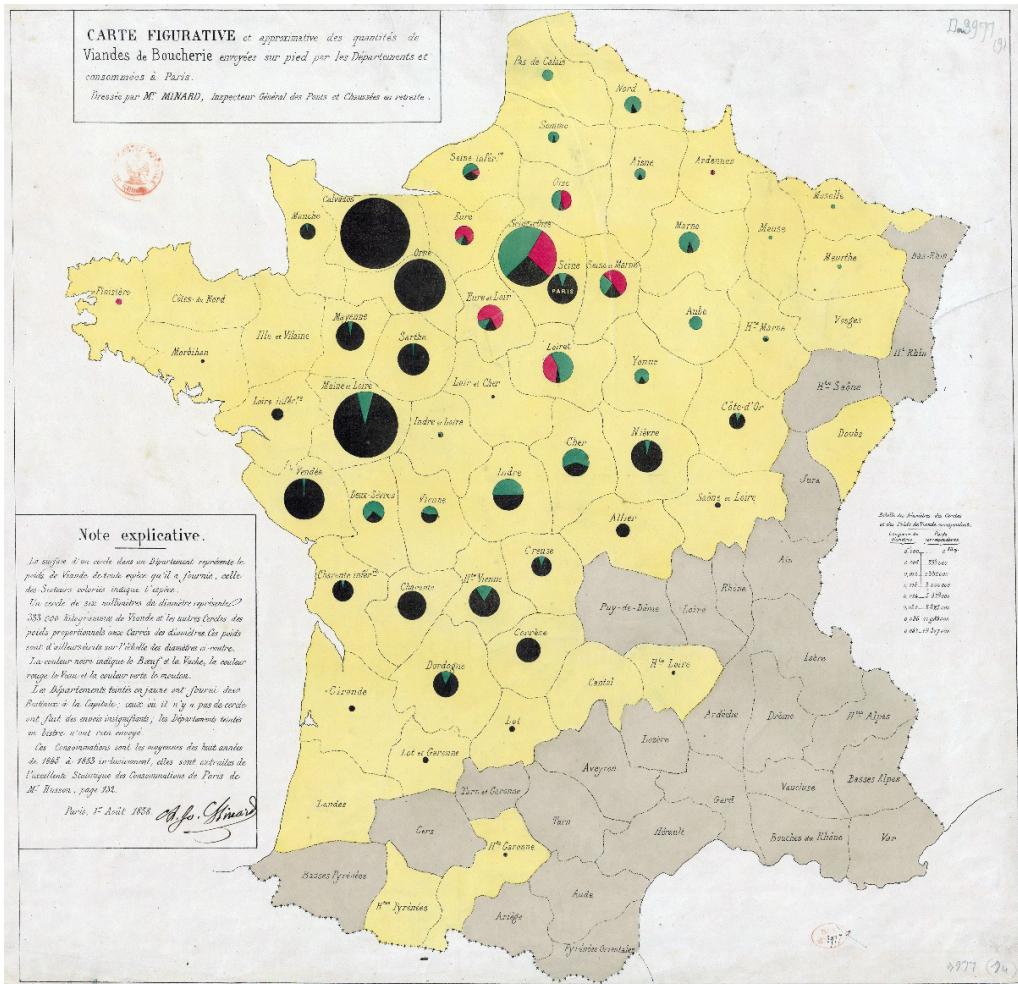


Figure 1.5 – Another map by Minard combining geography and statistics showing beef production in France using pie charts

In the early 1900s, mass printing drove the development of the concept of map layers – a key feature of geospatial analysis. Cartographers drew different map elements (vegetation, roads, and elevation contours) on plates of glass that could then be stacked and photographed to be printed as a single image. If the cartographer made a mistake, only one plate of glass had to be changed instead of the entire map. Later, the development of plastic sheets made it even easier to create, edit, and store maps in this manner. However, the layering concept for maps as a benefit to analysis would not come into play until the modern computer age.

## Evolution of Geographic Information Systems (GISs)

Computer mapping evolved with the computer itself in the 1960s. However, the origin of the term *GIS* began with the Canadian Department of Forestry and Rural Development. Dr. Roger Tomlinson headed a team of 40 developers in an agreement with IBM to build the **Canada Geographic Information System (CGIS)**. The CGIS tracked the natural resources of Canada and allowed those features to be profiled for further analysis. The CGIS stored each type of land cover as a different layer.

It also stored data in a Canadian-specific coordinate system, suitable for the entire country, which was devised for optimal area calculations. While the technology that was used was primitive by today's standards, the system had phenomenal capability at that time. The CGIS included the following software features, all of which can still be found in modern GIS software over 60 years later:

- Map projection switching
- The rubber sheeting of scanned images
- Map scale change
- Line smoothing and generalization to reduce the number of points in a feature
- Automatic gap closing for polygons
- Area measurement
- The dissolving and merging of polygons
- Geometric buffering
- The creation of new polygons
- Scanning
- The digitizing of new features from the reference data

### More about CGIS

The National Film Board of Canada produced a documentary in 1967 on the CGIS, which you can view at <https://www.youtube.com/watch?v=3VLGvWEuZxI>.

Tomlinson was often called the father of GIS. After launching the CGIS, he earned his doctorate from the University of London with his 1974 dissertation, titled *The application of electronic computing methods and techniques to the storage, compilation, and assessment of mapped data*, which describes GIS and geospatial analysis. Tomlinson ran his own global consulting firm, Tomlinson Associates Ltd., where he remained an active participant in the industry late in life. He was often found delivering keynote addresses at geospatial conferences.

CGIS is the starting point of geospatial analysis, as defined by this book. However, this book would not have been written if not for the work of Howard Fisher and the Harvard Laboratory for Computer Graphics and Spatial Analysis at the Harvard Graduate School of Design. His work on the SYMAP GIS software, which outputs maps to a line printer, started an era of development at the laboratory, which produced two other important packages and, as a whole, permanently defined the geospatial industry. SYMAP led to other packages, including GRID and the Odyssey project, which come from the same lab:

- GRID was a raster-based GIS system that used cells to represent geographic features instead of geometry. GRID was written by Carl Steinitz and David Sinton. The system later became IMGRID.
- Odyssey was a team effort led by Nick Chrisman and Denis White. It was a system of programs that included many advanced geospatial data management features that are typical of modern geodatabase systems. Harvard attempted to commercialize these packages with limited success. However, their impact is still seen today.

Virtually, every existing commercial and open source package owes something to these code bases.

Howard Fisher produced a film in 1967 using the output from SYMAP to show the urban expansion of Lansing, Michigan, from 1850 to 1965 by hand-coding decades of property information into the system. This analysis took months. However, in this day and age, it would take only a few minutes to recreate them because of modern tools and data.

#### More on SYMAP

You can watch the film at [https://www.youtube.com/watch?v=xj8DQ7IQ8\\_o](https://www.youtube.com/watch?v=xj8DQ7IQ8_o).

There are dozens of **graphical user interface (GUI)** geospatial desktop applications available today from companies, including Esri, ERDAS, Intergraph, ENVI, and so on. Esri is the oldest, continuously operating GIS software company, which started in the late 1960s. In the open source realm, packages including **Quantum GIS (QGIS)** and the **Geographic Resources Analysis Support System (GRASS)** are widely used. Beyond comprehensive desktop software packages, software libraries for building new software exist in the thousands.

GIS can provide detailed information about the Earth, but it is still just a model. Sometimes, we need a direct representation to gain knowledge about current or recent changes on our planet. At that point, we need remote sensing.

## Remote sensing

Remote sensing is where you collect information about an object without making physical contact with that object. In the context of geospatial analysis, that object is usually the Earth. Remote sensing also includes processing the collected information. The potential of geographic information systems

is limited only by the available geographic data. The cost of land surveying, even using a modern GPS to populate a GIS, has always been resource-intensive.

The advent of remote sensing not only dramatically reduced the cost of geospatial analysis but took the field in entirely new directions. In addition to powerful reference data for GIS systems, remote sensing has made generating automated and semi-automated GIS data possible by extracting features from images and geographic data. The eccentric French photographer Gaspard-Félix Tournachon, also known as Nadar, took the first aerial photograph in 1858, from a hot air balloon over Paris:

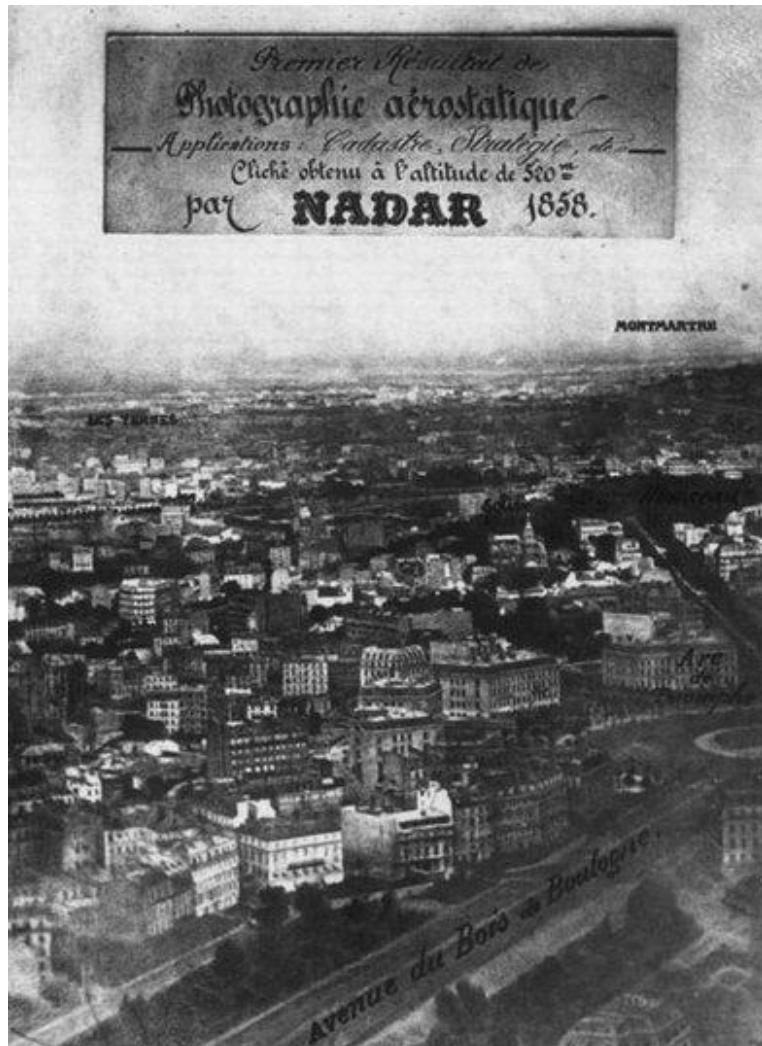


Figure 1.6 – An aerial photo of Paris from a hot air balloon taken in 1858 by Nadar. It is considered to be the first aerial photo and the dawn of geospatial remote sensing

The value of a true bird's-eye view of the world was immediately apparent. As early as 1920, books on aerial photo interpretation began to appear.

When the United States entered the Cold War with the Soviet Union after World War II, aerial photography to monitor military capability became prolific with the invention of the American U-2 spy plane. The U-2 spy plane could fly at 75,000 feet, putting it out of the range of existing anti-aircraft weapons designed to reach only 50,000 feet. The American U-2 flights over Russia ended when the Soviets finally shot down a U-2 and captured the pilot.

However, aerial photography had little impact on modern geospatial analysis. Planes could only capture small footprints of an area. Photographs were tacked to walls or examined on light tables but not in the context of other information. Though extremely useful, aerial photo interpretation was simply another visual perspective.

The game changer came on October 4, 1957, when the Soviet Union launched the Sputnik 1 satellite. The Soviets had scrapped a much more complex and sophisticated satellite prototype because of manufacturing difficulties. Once corrected, this prototype would later become Sputnik 3. Instead, they opted for a simple metal sphere with four antennae and a simple radio transmitter. Other countries, including the United States, were also working on satellites. These satellite initiatives were not entirely a secret. They were driven by scientific motives as part of the **International Geophysical Year (IGY)**.

Advancements in rocket technology made artificial satellites a natural evolution for Earth science. However, in nearly every case, each country's defense agency was also heavily involved. Similar to the Soviets, other countries were struggling with complex satellite designs packed with scientific instruments. The Soviets' decision to switch to the simplest possible device was for the sole reason of launching a satellite before the Americans were effective. Sputnik was visible in the sky as it passed over, and its radio pulse could be heard by amateur radio operators. Despite Sputnik's simplicity, it provided valuable scientific information that could be derived from its orbital mechanics and radiofrequency physics.

The Sputnik program's biggest impact was on the American space program. America's chief adversary had gained a tremendous advantage in the space race. The United States ultimately responded with the Apollo moon landings. However, before this, the US launched a program that would remain a national secret until 1995. The classified CORONA program resulted in the first pictures from space. The US and Soviet Union had signed an agreement to end spy plane flights, but satellites were conspicuously absent from the negotiations.

The following figure shows the CORONA process. The dashed lines are the satellite flight paths, the long white tubes are the satellites, the small white cones are the film canisters, and the black blobs are the control stations that triggered the ejection of the film so that a plane could catch it in the sky:

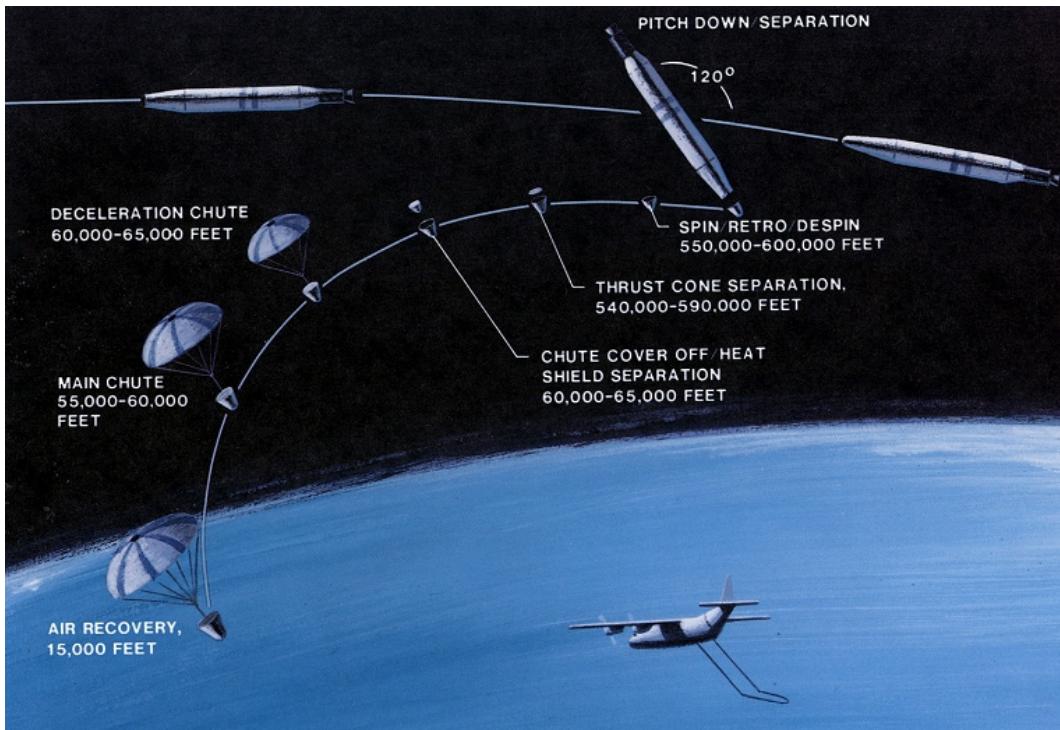


Figure 1.7 – An illustration of the early CORONA spy satellite that ejected film canisters that were caught in mid-air by a plane

The first CORONA satellite was a four-year effort with many setbacks. However, the program ultimately succeeded. The difficulty with satellite imaging, even today, is retrieving the images from space. The CORONA satellites used canisters of black and white film that were ejected from the vehicle once exposed. As a film canister parachuted to Earth, a US military plane would catch the package in midair. If the plane missed the canister, it would float for a brief period in the water before sinking into the ocean to protect the sensitive information.

The US continued to develop the CORONA satellites until they matched the resolution and photographic quality of the U-2 spy plane photos. The primary disadvantages of the CORONA instruments were reusability and timeliness. Once out of film, a satellite would no longer be of service. Additionally, the film's recovery was on a set schedule, making the system unsuitable for monitoring real-time situations. The overall success of the CORONA program, however, paved the way for the next wave of satellites, which ushered in the modern era of remote sensing.

Due to the CORONA program's secret status, its impact on remote sensing was indirect. Photographs of the Earth taken on manned US space missions inspired the idea of a civilian-operated remote-sensing satellite. The benefits of such a satellite were clear, but the idea was still controversial. Government officials questioned whether a satellite was as cost-efficient as aerial photography. The military was

worried that the public satellite could endanger the secrecy of the CORONA program. Other officials worried about the political consequences of imaging other countries without permission. However, the **Department of the Interior (DOI)** finally won permission for NASA to create a satellite to monitor Earth's surface resources.

On July 23, 1972, NASA launched the **Earth Resources Technology Satellite (ERTS)**. The ERTS was quickly renamed Landsat 1. The platform contained two sensors. The first was the **Return Beam Vidicon (RBV)** sensor, which was essentially a video camera. It was built by the radio and television giant known as the **Radio Corporation of America (RCA)**. The RBV immediately had problems, which included disabling the satellite's altitude guidance system. The second sensor was the highly experimental **Multispectral Scanner (MSS)**. The MSS performed flawlessly and produced superior results compared to the RBV. The MSS captured four separate images at four different wavelengths of the light reflected from the Earth's surface.

This sensor had several revolutionary capabilities. The first and most important capability was the first global imaging of the planet scanning every spot on Earth every 16 days. It also recorded light beyond the visible spectrum. While it did capture green and red light visible to the human eye, it also scanned near-infrared light at two different wavelengths not visible to the human eye. The images were stored and transmitted digitally to three different ground stations in Maryland, California, and Alaska. Its multispectral capabilities and digital format meant that the aerial view provided by Landsat wasn't just another photograph from the sky. It was beaming down the data. This data could be processed by computers to output derivative information about the Earth in the same way a GIS provided derivative information about the Earth by analyzing one geographic feature in the context of another. NASA promoted the use of Landsat worldwide and made the data available at very affordable prices to anyone who asked.

This global imaging capability led to many scientific breakthroughs, including the discovery of previously unknown geography, which occurred as late as 1976. For example, using Landsat imagery, the government of Canada located a tiny uncharted island inhabited by polar bears. They named the new landmass Landsat Island.

Landsat 1 was followed by six other missions that were turned over to the **National Oceanic and Atmospheric Administration (NOAA)** as the responsible agency. Landsat 6 failed to achieve orbit due to a ruptured manifold, which disabled its maneuvering engines. During some of these missions, the satellites were managed by the **Earth Observation Satellite (EOSAT)** company, now called **Space Imaging**, but returned to government management by the Landsat 7 mission. The following figure from NASA is a sample of a Landsat 7 product over Cape Cod, Massachusetts, USA:



Figure 1.8 – An example of a Landsat 7 satellite image over Cape Cod, Massachusetts, USA

The **Landsat Data Continuity Mission (LDCM)** was launched on February 13, 2013, and began collecting images on April 27, 2013, as part of its calibration cycle to become Landsat 8. The LDCM is a joint mission between NASA and the **US Geological Survey (USGS)**.

## Point cloud data

Remote sensing data can measure the Earth in two dimensions. But we can also use remote sensing to measure the Earth and things on it in three dimensions using point cloud data. **Point cloud data** are collections of discrete points with horizontal coordinates represented by X and Y values and a vertical coordinate represented by a Z value. These points are collected in a variety of ways, including lasers (as in the case of lidar data), sound (which is commonly used for mapping the seafloor but can also be used on land), stereoscopic imagery, radio waves (such as radar), or “structure-in-motion” data, where a single geo-located camera is used to collect overlapping images to estimate the 3D structure of a scene. Point cloud data can be mapped relative to any origin and is not necessarily dependent on

GPS like other types of geospatial data. This feature allows it to be used to map indoor spaces, as well as outdoor spaces. Point cloud data can also be colorized for visualization using color photography or video for creating immersive 3D models, as shown in the following colorized point cloud of Red Rocks, Colorado, USA:

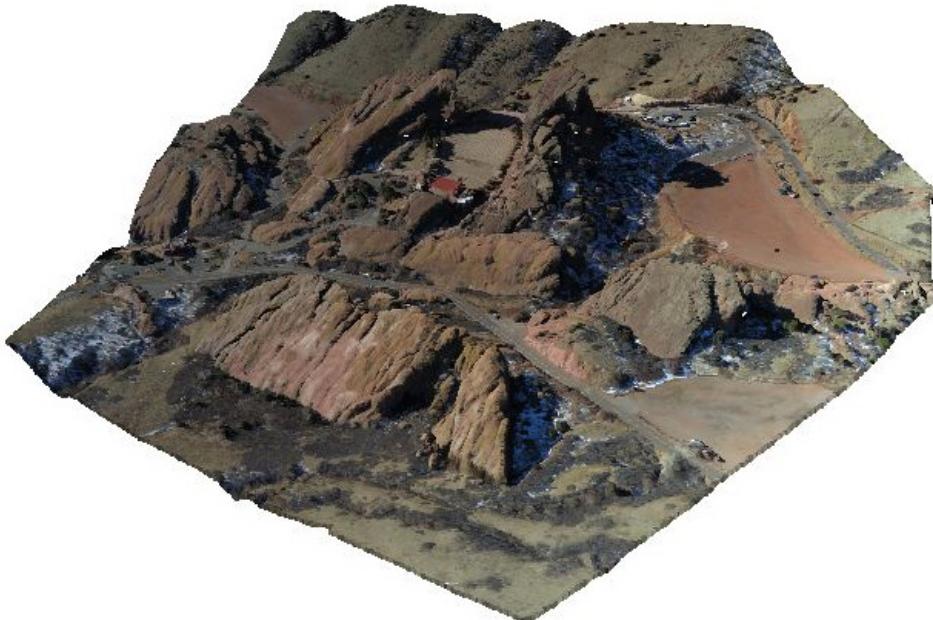


Figure 1.9 – A colorized lidar point cloud of Red Rocks, Colorado, USA, which creates a photorealistic 3D model

Point cloud data is the most common way to create elevation data that can be turned into a gridded format. This is called a **Digital Elevation Model (DEM)**. A DEM is a three-dimensional representation of a planet's terrain. In the context of this book, this planet is the Earth. The history of digital elevation models is far less complicated than remotely-sensed imagery but no less significant. Before computers, representations of elevation data were limited to topographic maps created through traditional land surveys. The technology existed to create three-dimensional models from stereoscopic images or physical models from materials such as clay or wood, but these approaches were not widely used for geography.

The concept of digital elevation models came about in 1986 when the French space agency, **Centre National d'études Spatiales (CNES)** or National Center for the Study of Space, launched its SPOT-1 satellite, which included a stereoscopic radar. This system created the first usable DEM. Several other US and European satellites followed this model with similar missions.

In February 2000, the space shuttle Endeavour conducted the **Shuttle Radar Topography Mission (SRTM)**, which collected elevation data of over 80% of the Earth's surface using a special radar antenna

configuration that allowed a single pass. This model was surpassed in 2009 by the joint US and Japanese mission, which used the **Advanced Spaceborne Thermal Emission and Reflection Radiometer** (**ASTER**) sensor aboard NASA's Terra satellite. This system captured 99% of the Earth's surface but has proven to have minor data issues. Since the Space Shuttle's orbit did not cross the Earth's poles, it did not capture the entire surface. SRTM remains the gold standard. The following figure from the USGS (<https://www.usgs.gov/media/images/national-elevation-dataset>) shows a colorized DEM known as a hill shade. Greener areas are at lower elevations, while yellow and brown areas are at mid-range to high elevations:

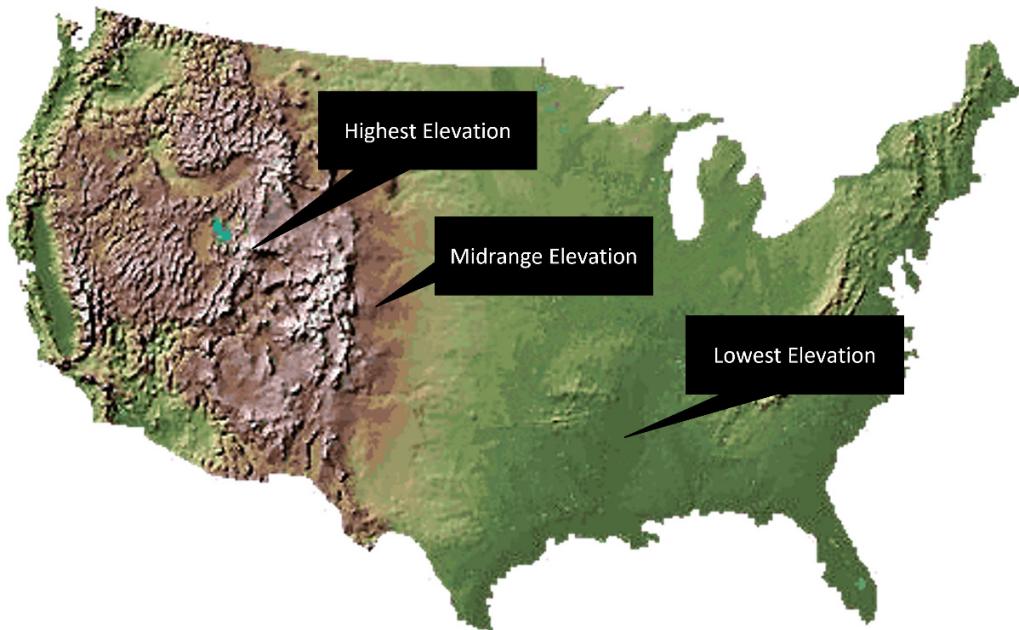


Figure 1.10 – An example of a DEM using colors and hill shade shadows to create a natural-looking representation of elevation

Recently, more ambitious attempts at a worldwide elevation dataset are underway in the form of the TerraSAR-X and TanDEM-X satellites, which were launched by Germany in 2007 and 2010, respectively. These two radar elevation satellites worked together to produce a global DEM called WorldDEM, which was released on April 15, 2014. This dataset has a relative accuracy of 2 meters and an absolute accuracy of 4 meters.

## Computer-aided drafting

**Computer-aided drafting (CAD)** is worth mentioning, though it does not directly relate to geospatial analysis. The history of CAD system development parallels and intertwines with the history of geospatial analysis. CAD is an engineering tool that's used to model two- and three-dimensional objects, usually for engineering and manufacturing purposes. The primary difference between a geospatial model and a CAD model is that a geospatial model is referenced to the Earth, whereas a CAD model can exist in abstract space.

For example, a three-dimensional blueprint of a building in a CAD system would not have latitude or longitude, but in a GIS, the same building model would have a location on Earth. However, over the years, CAD systems have taken on many features of GIS systems and are commonly used for smaller GIS projects. Likewise, many GIS programs can import CAD data that has been georeferenced. Traditionally, CAD tools were designed primarily to engineer data that was not geospatial.

However, engineers who became involved with geospatial engineering projects, such as designing a city's utility electric system, would use the CAD tools that they were familiar with to create maps. Over time, the GIS software evolved to import the geospatial-oriented CAD data produced by engineers, and CAD tools evolved to support geospatial data creation and better compatibility with GIS software. AutoCAD by Autodesk and ArcGIS by Esri were the leading commercial packages to develop this capability, and the **Geospatial Data Abstraction Library (GDAL)** OGR library developers added CAD support as well.

## Geospatial analysis and computer programming

Modern geospatial analysis can be conducted with the click of a button in any of the easy-to-use commercial or open source geospatial packages. So, why would you want to use a programming language to learn this field? The most important reasons are as follows:

- You want complete control of the underlying algorithms, data, and execution
- You want to automate task-specific, repetitive analysis tasks with minimal overhead from a large, multipurpose geospatial framework
- You want to create a program that's easy to share
- You want to learn geospatial analysis beyond pushing buttons in software

The geospatial industry is gradually moving away from the traditional workflow, in which teams of analysts use expensive desktop software to produce geospatial products.

Geospatial analysis is being pushed toward automated processes that reside in the cloud. End user software is moving toward task-specific tools, many of which are accessed from mobile devices. Knowledge of geospatial concepts and data, as well as the ability to build custom geospatial processes, is where the geospatial work in the future lies.

## Object-oriented programming for geospatial analysis

Object-oriented programming is a software development paradigm in which concepts are modeled as objects that have properties and behaviors represented as attributes and methods, respectively. The goal of this paradigm is more modular software in which one object can inherit from one or more other objects to encourage software reuse.

The Python programming language is known for its ability to serve multiple roles as a well-designed, object-oriented language, a procedural scripting language, or even a functional programming language. However, you never completely abandon object-oriented programming in Python because even its native data types are objects and all Python libraries, known as modules, adhere to a basic object structure and behavior.

Geospatial analysis is the perfect activity for object-oriented programming. In most object-oriented programming projects, the objects are abstract concepts, such as database connections that have no real-world analogy. However, in geospatial analysis, the concepts that are modeled are, well, real-world objects! The domain of geospatial analysis is the Earth and everything on it. Trees, buildings, rivers, and people are all examples of objects within a geospatial system.

A common example in literature for newcomers to object-oriented programming is the concrete analogy of a cat. Books on object-oriented programming frequently use some form of the following example.

Imagine that you are looking at a cat. We know some information about the cat, such as its name, age, color, and size. These features are the properties of the cat. The cat also exhibits behaviors such as eating, sleeping, jumping, and purring. In object-oriented programming, objects have properties and behaviors too. You can model a real-world object such as the cat in our example, or something more abstract such as a bank account.

Most concepts in object-oriented programming are far more abstract than the simple cat paradigm or even a bank account. However, in geospatial analysis, the objects that are modeled remain concrete, such as the simple cat analogy, and in many cases are cats.

Geospatial analysis allows you to continue with the simple cat analogy and even visualize it. The following figure represents the feral cat population of Australia using data provided by the **Atlas of Living Australia (ALA)**:



Figure 1.11 – A geospatial heat map of feral cat populations in Australia illustrating that in object-oriented geospatial programming, objects are real-world objects and not just software abstractions

So, we can use computers to analyze the relationships between features on Earth, but why should we? In the next section, we'll look at why geospatial analysis is a worthwhile endeavor.

## The importance of geospatial analysis

Geospatial analysis helps people make better decisions. It doesn't make the decision for you, but it can answer critical questions that are at the heart of the choice to be made and often cannot be answered any other way. Until recently, geospatial technology and data were tools available only to governments and well-funded researchers. However, in the last decade, data has become much more widely available and software has become much more accessible to anyone.

In addition to freely available government satellite imagery, many local governments now conduct aerial photo surveys and make the data available online. The ubiquitous Google Earth provides a cross-platform spinning globe view of the Earth with satellite and aerial data, streets, points of interest, photographs, and much more. Google Earth users can create custom **Keyhole Markup Language (KML)** files, which are XML files that are used to load and style data to the globe. This program and similar tools are often called geographic exploration tools because they are excellent data viewers but provide very limited data analysis capabilities.

The ambitious OpenStreetMap project (<https://www.openstreetmap.org/#map=5/51.500/-0.100>) is a crowd-sourced, worldwide, geographic base map containing most layers commonly found in a GIS. Nearly every mobile phone now contains a GPS, along with mobile apps to collect GPS tracks as points, lines, or polygons. Most phones will also tag photos taken with the phone's camera with GPS coordinates. In short, anyone can be a geospatial analyst.

The global population has reached 7 billion people. The world is changing faster than ever before. The planet is undergoing environmental changes that have never been seen in recorded history. Faster communication and transportation increase the interaction between us and the environment in which we live. Managing people and resources safely and responsibly is more challenging than ever. Geospatial analysis is the best approach to understanding our world more efficiently and deeply. The more politicians, activists, relief workers, parents, teachers, first responders, medical professionals, and small businesses that harness the power of geospatial analysis, the more potential we have for a better, healthier, safer, and fairer world.

## GIS concepts

To begin geospatial analysis, we need to understand some key underlying concepts that are unique to the field. The list isn't long, but nearly every aspect of analysis traces back to one of these ideas.

### Thematic maps

As its name suggests, a thematic map portrays a specific theme. A general reference map visually represents features as they relate geographically to navigation or planning. A thematic map goes beyond location to provide the geographic context for information around a central idea. Usually, a thematic map is designed for a targeted audience to answer specific questions. The value of thematic maps lies in what they do not show. A thematic map uses minimal geographic features to avoid distracting the reader from the theme. Most thematic maps include political boundaries such as country or state borders but omit navigational features, such as street names or points of interest beyond major landmarks that orient the reader.

The cholera map by Dr. John Snow, which was provided earlier in this chapter, is a perfect example of a thematic map. Common uses for thematic maps include visualizing health issues, such as disease, election results, and environmental phenomena such as rainfall. These maps are also the most common output of geospatial analysis. The following map from the United States Census Bureau shows monthly business applications by state:

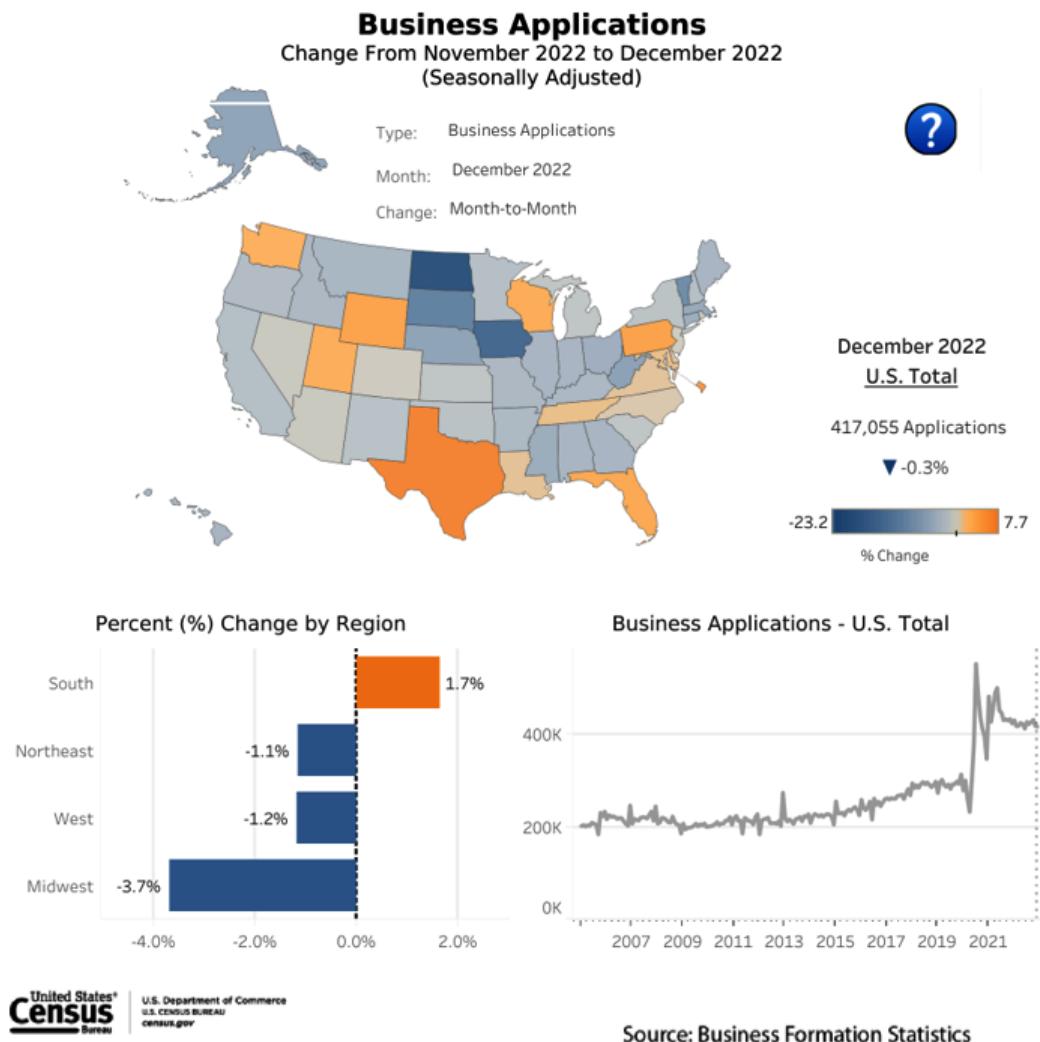


Figure 1.12 – A modern example of a thematic map from the US Census Bureau showing the distribution of business applications by states

Thematic maps tell a story and are very useful. However, it is important to remember that, while thematic maps are models of reality just like any other map, they are also generalizations of information. Two different analysts using the same source of information will often come up with very different thematic maps, depending on how they analyze and summarize the data. They may also choose to focus on different aspects of the dataset. The technical nature of thematic maps often leads people to treat them as if they are scientific evidence. However, geospatial analysis is often inconclusive. While the analysis may be based on scientific data, the analyst does not always follow the rigor of the scientific method.

In his classic book, *How to Lie with Maps*, Mark Monmonier, University of Chicago Press, demonstrates in detail how maps are easily manipulated models of reality, which are commonly abused. This fact doesn't degrade the value of these tools. The legendary statistician, George Box, wrote the following in his 1987 book, *Empirical Model-Building and Response Surfaces*:

“Essentially, all models are wrong, but some are useful.”

Thematic maps have been used as guides to start (and end) wars, stop deadly diseases in their tracks, win elections, feed nations, fight poverty, protect endangered species, and rescue those impacted by a disaster. Thematic maps may be the most useful models ever created.

## Spatial databases

In its purest form, a database is simply an organized collection of information. A **database management system (DBMS)** is an interactive suite of software that can interact with a database. People often use the word database as a catch-all term that refers to both the DBMS and the underlying data structure. Databases typically contain alphanumeric data and, in some cases, **binary large objects (blobs)**, which can store binary data such as images. Most databases also allow a relational database structure in which entries in normalized tables can be referenced to each other to create many-to-one and one-to-many relationships among data.

Spatial databases, also known as geodatabases, use specialized software to extend a traditional **relational database management system (RDBMS)** to store and query data defined in a two- or three-dimensional space. Some systems also account for a series of data over time. In a spatial database, attributes about geographic features are stored and queried as traditional relational database structures. These spatial extensions allow you to query geometries using **Structured Query Language (SQL)** in a similar way to traditional database queries. Spatial queries and attribute queries can also be combined to aid with selecting results based on both location and attributes.

## Spatial indexing

Spatial indexing is a process that organizes geospatial vector data for faster retrieval. It is a way of prefiltering the data for common queries or rendering. Indexing is commonly used in large databases to speed up the returns to queries. Spatial data is no different. Even a moderately sized geodatabase can contain millions of points or objects. If you perform a spatial query, every point in the database must be considered by the system for it to include them or eliminate them from the results. Spatial indexing groups data in ways that allow large portions of the dataset to be eliminated from consideration by doing computationally simpler checks before going into a detailed and slower analysis of the remaining items.

## Metadata

Metadata is defined as data about data. Accordingly, geospatial metadata is data about geospatial datasets that provides traceability for the source and history of a dataset, as well as a summary of the

technical details. Metadata also provides long-term preservation of data by way of documenting the asset over time.

Geospatial metadata can be represented by several possible standards. One of the most prominent standards is the international standard, ISO 19115-1, which includes hundreds of potential fields to describe a single geospatial dataset. Additionally, the ISO 19115-2 standard includes extensions for geospatial imagery and gridded data. Some example fields include spatial representation, temporal extent, and lineage. ISO 19115-3 is the standard for describing the procedure to generate an XML schema from ISO geographic metadata. Dublin Core is another international standard that was developed for digital data that has been extended for geospatial data, along with the associated DCAT vocabulary for building catalogs of data from a single source.

The primary use of metadata is for cataloging datasets. Modern metadata can be ingested by geographic search engines, making it potentially discoverable by other systems automatically. It also lists points of contact for a dataset if you have questions.

### Python and metadata

Metadata is an important support tool for geospatial analysts and adds credibility and accessibility to your work. The **Open Geospatial Consortium (OGC)**, which created the **Catalog Service for the Web (CSW)**, is used to manage metadata. The `pycsw` Python library implements the CSW standard.

## Map projections

Map projections have entire books devoted to them and can be a challenge for new analysts. If you take any 3D object and flatten it on a plane, such as your screen or a sheet of paper, the object will be distorted. Many grade school geography classes demonstrate this concept by having students peel an orange and then attempt to lay the peel flat on their desks to understand the resulting distortion. The same effect occurs when you take the round shape of the Earth and project it onto a computer screen.

In geospatial analysis, you can manipulate this distortion to preserve common properties, such as area, scale, bearing, distance, or shape. There is no one-size-fits-all solution to map projections. The choice of projection is always a compromise of gaining accuracy in one dimension in exchange for errors in another. Projections are typically represented as a set of over 40 parameters, either in XML or in a text format called **Well-Known Text (WKT)**, which is used to define the transformation algorithm.

The **International Association of Oil and Gas Producers (IOGP)** maintains a registry of the most well-known projections. The organization was formerly known as the **European Petroleum Survey Group (EPSG)**. The entries in the registry are still known as EPSG codes. The EPSG maintained the registry as a common benefit for the oil and gas industry, which is a prolific user of geospatial analysis for energy exploration. At the last count, this registry contained over 5,000 entries.

As recently as 10 years ago, map projections were of primary concern for a geospatial analyst. Data storage was expensive, high-speed internet was rare, and cloud computing didn't really exist. Geospatial data was typically exchanged among small groups working in separate areas of interest. The technology constraints at the time meant that geospatial analysis was highly localized. Analysts would use the best projection for their area of interest.

Data in different projections could not be displayed on the same map because they represent two different models of the Earth. Any time an analyst received data from a third party, it had to be reprojected before they could use it with the existing data. This process was tedious and time-consuming.

Most geospatial data formats do not provide a way to store the projection information. This information is stored in an ancillary file, usually as text or XML. Since analysts didn't exchange data often, many people wouldn't bother defining projection information. Every analyst's nightmare was to come across an extremely valuable dataset that was missing the projection information. It rendered the dataset useless. The coordinates in the file are just numbers and offer no clue about the projection. With over 5,000 choices, it was nearly impossible to guess.

Now, thanks to modern software and the internet making data exchange easier and more common, nearly every data format has added a metadata format that defines a projection or places it in the file header, if supported. Advances in technology have also allowed for global base maps, which allow for more common uses of projections, such as the common Google Mercator projection, which is used for Google Maps. This projection is also known as Web Mercator and uses code **EPSG:3857** (or the deprecated EPSG:900913).

Geospatial portal projects such as OpenStreetMap (<https://www.openstreetmap.org>) have consolidated datasets for much of the world in common projections. Modern geospatial software can also reproject data on the fly, saving the analyst the trouble of preprocessing the data before using it. Closely related to map projections are geodetic datums. A datum is a model of the Earth's surface that's used to match the location of features on the Earth to a coordinate system. One common datum is called WGS 84 and is used by GPS.

## Rendering

The exciting part of geospatial analysis is visualization. Since geospatial analysis is a computer-based process, it is good to be aware of how geographic data appears on a computer screen.

Geographic data including points, lines, and polygons are stored numerically as one or more points, which come in  $(X, Y)$  pairs or  $(X, Y, Z)$  tuples. The  $X$  represents the horizontal axis on a graph, while the  $Y$  represents the vertical axis. The  $Z$  represents terrain elevation. In computer graphics, a computer screen is represented by an  $X$ - and  $Y$ -axis. The  $Z$ -axis is not used because the computer screen is treated as a two-dimensional plane by most graphics software APIs. However, because desktop computing power continues to improve, three-dimensional maps are starting to become more common.

Another important factor is screen coordinates versus world coordinates. Geographic data is stored in a coordinate system representing a grid overlaid on the Earth, which is three-dimensional and

round. Screen coordinates, also known as pixel coordinates, represent a grid of pixels on a flat, two-dimensional computer screen. Mapping X and Y world coordinates to pixel coordinates is fairly straightforward and involves a simple scaling algorithm. However, if a Z coordinate exists, then a more complicated transformation must be performed to map coordinates from a three-dimensional space to a two-dimensional plane. These transformations can be computationally costly and therefore slow if not handled correctly.

In the case of remote sensing data, the challenge is typically the file size. Even a moderately sized satellite image that is compressed can be tens, if not hundreds, of megabytes. Images can be compressed using two methods:

- **Lossless methods:** They use tricks to reduce the file size without discarding any data
- **Lossy compression algorithms:** They reduce the file size by reducing the amount of data in the image while avoiding a significant change in the appearance of the image

Rendering an image on the screen can be computationally intensive. Most remote sensing file formats allow you to store multiple lower-resolution versions of the image – called overviews or pyramids – for the sole purpose of faster rendering at different scales. When zoomed out from the image to a scale where you can see the detail of the full-resolution image, a preprocessed, lower-resolution version of the image is displayed quickly and seamlessly.

## Remote sensing concepts

Most of the GIS concepts we've described also apply to raster data. However, raster data has some unique properties as well. Earlier in this chapter, when we went over the history of remote sensing, the focus was on Earth imaging from aerial platforms. It is important to note that raster data can come in many forms, including ground-based radar, laser range finders, and other specialized devices to detect gases, radiation, and other forms of energy in a geographic context.

For this book, we will focus on remote sensing platforms that capture large amounts of Earth data. These sources include Earth imaging systems, certain types of elevation data, and some weather systems, where applicable.

### Images as data

Raster data is captured digitally as square tiles. This means that the data is stored on a computer as a numerical array of rows and columns. If the data is multispectral, the dataset will usually contain multiple arrays of the same size, which are geospatially referenced together to represent a single area on the Earth. These different arrays are called bands.

Any numerical array can be represented on a computer as an image. In fact, all computer data is ultimately numbers. In geospatial analysis, it is important to think of images as a numeric array because mathematical formulas are used to process them.

In remotely sensed images, each pixel represents both space (the location on the Earth of a certain size) and the reflectance captured as light reflected from the Earth at that location into space. So, each pixel has a ground size and contains a number representing the intensity. Since each pixel is a number, we can perform mathematical equations on this data to combine data from different bands and highlight specific classes of objects in the image. If the wavelength value is beyond the visible spectrum, we can highlight features that aren't visible to the human eye. Substances such as chlorophyll in plants can be greatly contrasted using a specific formula called the **Normalized Difference Vegetation Index (NDVI)**.

By processing remotely sensed images, we can turn this data into visual information. Using the NDVI formula, we can answer the question, what is the relative health of the plants in this image? You can also create new types of digital information, which can be used as input for computer programs to output other types of information.

## Remote sensing and color

Computer screens display images as combinations of **Red, Green, and Blue (RGB)** to match the capability of the human eye. Satellites and other remote sensing imaging devices can capture light beyond this visible spectrum. On a computer, wavelengths beyond the visible spectrum are represented in the visible spectrum so that we can see them. These images are known as false color images. In remote sensing, for instance, infrared light makes moisture highly visible.

This phenomenon has a variety of uses, such as monitoring ground saturation during a flood or finding hidden leaks in a roof or levee.

## Common vector GIS concepts

In this section, we will discuss the different types of GIS processes that are commonly used in geospatial analysis. This list is not exhaustive; however, it will provide you with the essential operations that all other operations are based on. If you understand these operations, you will quickly understand much more complex processes as they are either derivatives or combinations of these processes.

### Data structures

GIS vector data uses coordinates consisting of, at a minimum, an X horizontal value and a Y vertical value to represent a location on Earth. In many cases, a point may also contain a Z value. Other ancillary values are possible, including measurements or timestamps.

These coordinates are used to form points, lines, and polygons to model real-world objects. Points can be a geometric feature in and of themselves or they can connect line segments. Closed areas created by line segments are considered polygons. Polygons model objects such as buildings, terrain, or political boundaries.

A GIS feature can consist of a single point, line, or polygon, or it can consist of more than one shape. For example, in a GIS polygon dataset containing world country boundaries, the Philippines, which is made up of 7,107 islands, would be represented as a single country made up of thousands of polygons.

Vector data typically represents topographic features better than raster data. Vector data has more accuracy potential and is more precise. However, collecting vector data on a large scale is also traditionally more costly than raster data.

Two other important terms related to vector data structures are **bounding box** and **convex hull**. The bounding box, or minimum bounding box, is the smallest possible square that contains all of the points in a dataset. The following diagram demonstrates a bounding box for a collection of points:

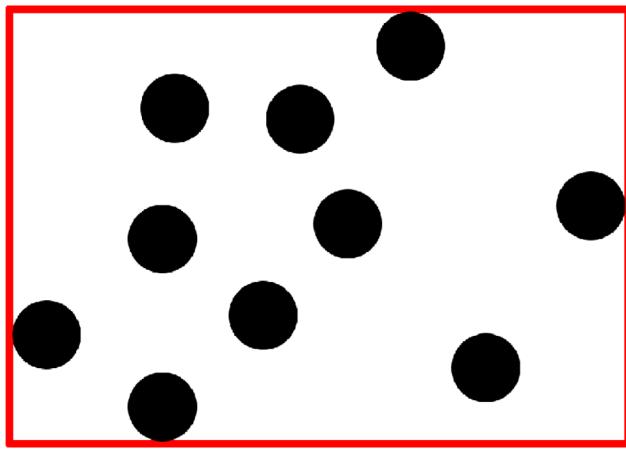


Figure 1.13 – A bounding box is the smallest possible box that fully contains a group of geospatial features

The convex hull of a dataset is similar to the bounding box, but instead of a square, it is the smallest possible polygon that can contain a dataset. The following diagram shows the same point data as the previous example, with the convex hull polygon shown in red:

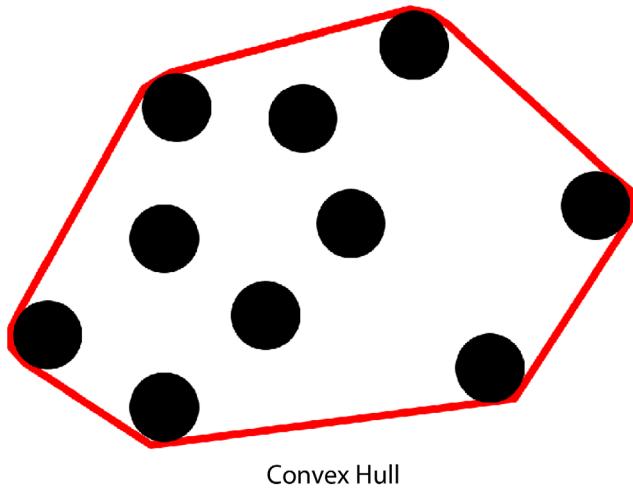


Figure 1.14 – A convex hull is the smallest possible polygon that fully contains a group of geospatial features

As you can see, the bounding box of a dataset always contains a convex hull.

### ***Geospatial rules about polygons***

In geospatial analysis, there are several general rules of thumb regarding polygons that are different from mathematical descriptions of polygons:

- Polygons must have at least four points – the first and last points must be the same
- A polygon boundary should not overlap itself
- Polygons in a layer shouldn't overlap
- A polygon in a layer inside another polygon is considered a hole in the underlying polygon

Different geospatial software packages and libraries handle exceptions to these rules differently, which can lead to confusing errors or software behaviors. The safest route is to make sure that your polygons obey these rules. There's one more important piece of information about polygons that we need to talk about.

A polygon is, by definition, a closed shape, which means that the first and last vertices of a polygon are identical. Some geospatial software will throw an error if you haven't explicitly duplicated the first point as the last point in the polygon dataset. Other software will automatically close the polygon without complaining. The data format that you use to store your geospatial data may also dictate how polygons are defined. This issue is a gray area, so it didn't make the polygon rules, but knowing this quirk will come in handy someday when you run into an error that you can't explain easily.

## Buffer

A buffer operation can be applied to spatial objects, including points, lines, or polygons. This operation creates a polygon around the object at a specified distance. Buffer operations are used for proximity analysis – for example, establishing a safety zone around a dangerous area. Let's review the following diagram:

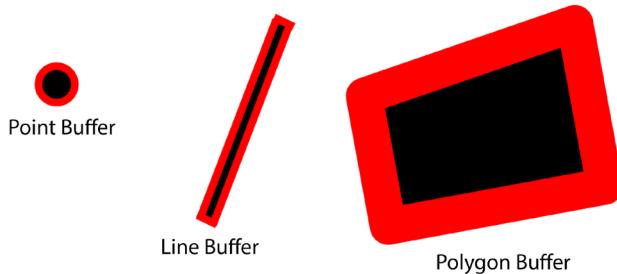
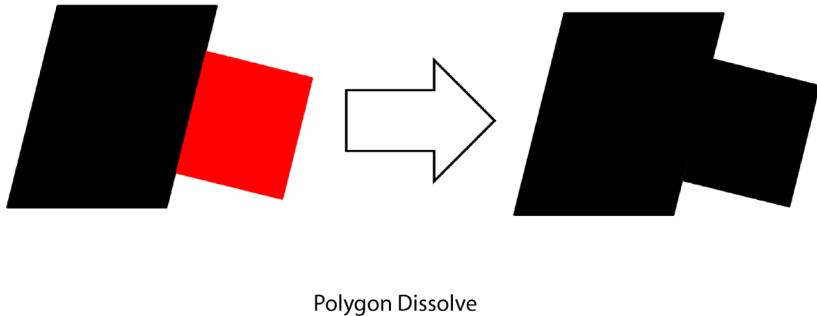


Figure 1.15 – A buffer is a polygon around a geospatial feature at a specified distance

The black shapes represent the original geometry, while the red outlines represent the larger buffer polygons that were generated from the original shape.

## Dissolve

A dissolve operation creates a single polygon out of adjacent polygons. Dissolves are also used to simplify data that's been extracted from remote sensing, as shown here:



Polygon Dissolve

Figure 1.16 – A polygon dissolve creates a single polygon out of adjacent polygons

A common use for a dissolve operation is to merge two adjacent properties in a tax database that has been purchased by a single owner.

## Generalize

Objects that have more points than necessary for the geospatial model can be generalized to reduce the number of points that are used to represent the shape. This operation usually requires a few attempts to get the optimal number of points without compromising the overall shape. It is a data optimization technique that's used to simplify data for the efficiency of computing or better visualization. This technique is useful in web mapping applications.

Here is an example of polygon generalization:

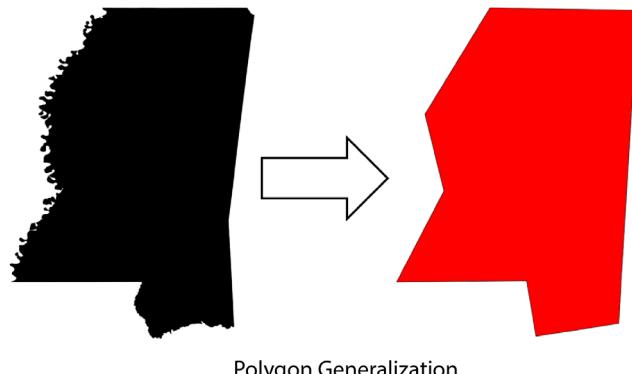
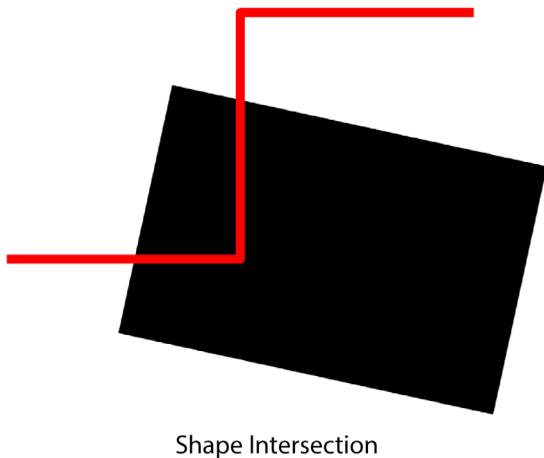


Figure 1.17 – Polygon generalization reduces the number of points in a polygon to simplify the geometry to speed up computation geometry or the graphical rendering of the feature. The compromise is losing detail in the shape, which may affect the visualization or analysis

Since computer screens have a resolution of **72 dots per inch (dpi)**, highly detailed point data, which would not be visible, can be reduced so that less bandwidth is used to send a visually equivalent map to the user.

## Intersection

An intersection operation is used to see if one part of a feature intersects with one or more features. This operation is used for spatial queries in proximity analysis and is often a follow-on operation to buffer analysis:

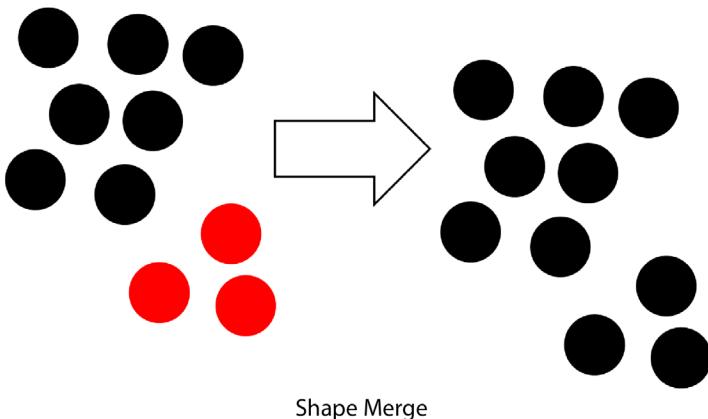


Shape Intersection

Figure 1.18 – A shape intersection checks whether one feature crosses the geometry of one or more other features

## Merge

A merge operation combines two or more non-overlapping shapes in a single multi-shape object. Multi-shape objects are shapes that maintain separate geometries but are treated as a single feature with a single set of attributes by the GIS:



Shape Merge

Figure 1.19 – A shape merge combines multiple non-overlapping features into a single dataset

## Point in polygon

A fundamental geospatial operation is checking to see whether a point is inside a polygon. This operation is the atomic building block of many different types of spatial queries. If the point is on the boundary of the polygon, it is considered inside. Very few spatial queries exist that do not rely on this calculation in some way. However, it can be very slow on a large number of points.

The most common and efficient algorithm to detect whether a point is inside a polygon is called the ray casting algorithm. First, a test is performed to see whether the point is on the polygon boundary. Next, the algorithm draws a line from the point in question in a single direction. The program counts the number of times the line crosses the polygon's boundary until it reaches the bounding box of the polygon, as shown here:

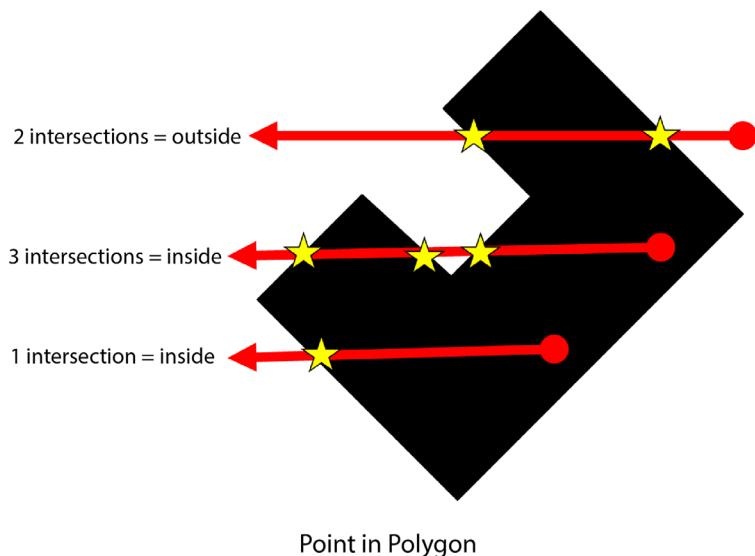
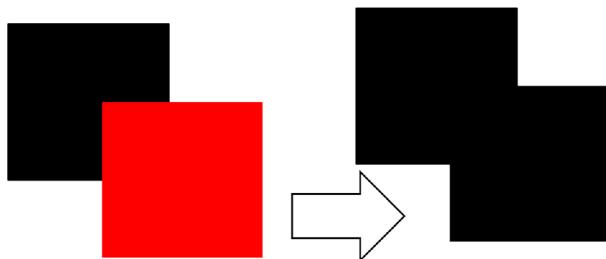


Figure 1.20 – The point-in-polygon ray casting algorithm is an efficient way to detect whether a point is inside a polygon

## Union

The union operation is less common but is very useful when you wish to combine two or more overlapping polygons in a single shape. It is similar to dissolve, but in this case, the polygons are overlapping as opposed to being adjacent:



Polygon Union

Figure 1.21 – A polygon union merges overlapping polygons into a single shape, similar to a dissolve, in which polygons are only adjacent

Usually, this operation is used to clean up automatically generated feature datasets from remote sensing operations.

## Join

A join or SQL join is a database operation that's used to combine two or more tables of information. Relational databases are designed to avoid storing redundant information for one-to-many relationships. For example, a US state may contain many cities. Rather than creating a table for each state containing all of its cities, a table of states with numeric IDs is created, while a table for all the cities in every state is created with a state numeric ID.

In a GIS, you can also have spatial joins that are part of the spatial extension software for a database. In spatial joins, you combine the attributes in the same way that you do in a SQL join. However, the relation is based on the spatial proximity of the two features.

To follow the previous cities example, we could add the county name that each city resides in using a spatial join. The cities layer could be loaded over a county polygon layer whose attributes contain the county's name. The spatial join would determine which city is in which county and perform a SQL join to add the county name to each city's attribute row.

## Common raster data concepts

As we mentioned earlier, remotely sensed raster data is a matrix of numbers. Remote sensing contains thousands of operations that can be performed on data. This field changes on almost a daily basis as new satellites are put into space and computer power increases.

Despite its decade-long history, we haven't even scratched the surface of the knowledge that this field can provide to the human race. Once again, similar to the common GIS processes, this minimal list of operations allows you to evaluate any technique that's used in remote sensing.

## Band math

Band math is multidimensional array mathematics. In array math, arrays are treated as single units, which are added, subtracted, multiplied, and divided. However, in an array, the corresponding numbers in each row and column across multiple arrays are computed simultaneously. These arrays are termed matrices, and computations involving matrices are the focus of linear algebra.

## Change detection

Change detection is the process of taking two images of the same location at different times and highlighting those changes. A change could be due to the addition of something on the ground, such as a new building, or the loss of a feature, such as coastal erosion. Many algorithms detect changes among images and also determine qualitative factors such as how long ago the change took place.

The following figure from a research project by the US **Oak Ridge National Laboratory (ORNL)** shows rainforest deforestation between 1984 and 2000 in the state of Rondonia, Brazil:

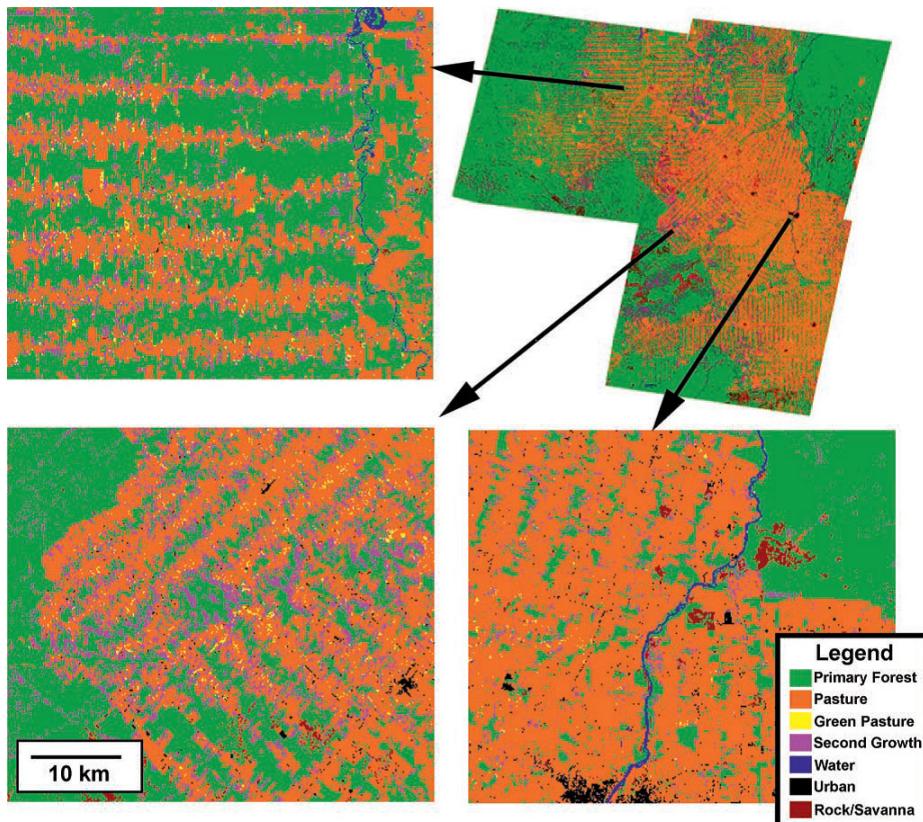


Figure 1.22 – This US Department of Energy satellite image analysis illustrates change detection by comparing the deforestation of a rainforest over time

Colors are used to show how recently the forest was cut. Green represents virgin rainforests, white represents a forest that was cut within 2 years of the end of the date range, red represents within 22 years, and the other colors fall in between, as described in the legend.

## Histogram

A histogram is the statistical distribution of values in a dataset. The horizontal axis represents a unique value in a dataset, while the vertical axis represents the frequency of this unique value in the raster. The following example, which was generated from a NASA Landsat image, shows a histogram showing pixel value distributions for the first three bands:

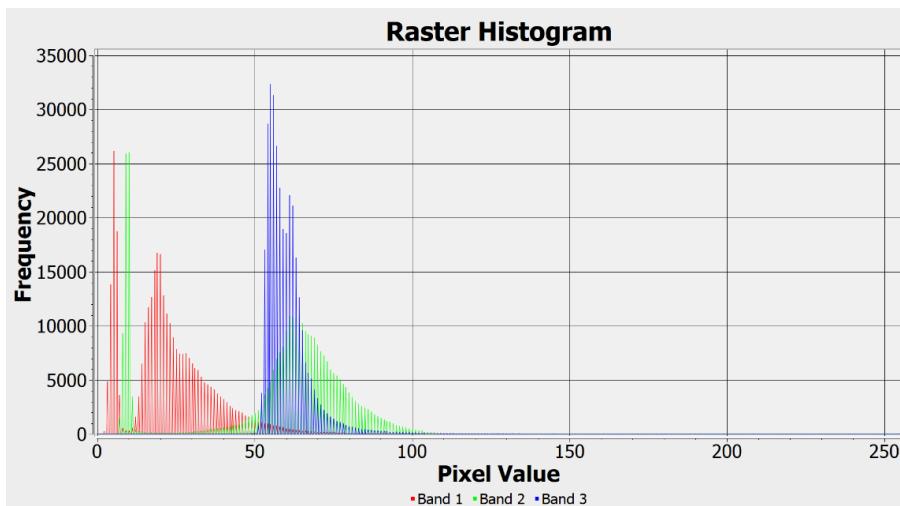


Figure 1.23 – Histogram distribution of red, green, and blue pixels in a satellite image that can be redistributed to enhance an image for visualizing or analyzing certain features

A histogram is a key operation in most raster processing. It can be used for everything from enhancing contrast in an image to serving as a basis for object classification and image comparison.

## Feature extraction

Feature extraction is the process of manually or automatically digitizing features in an image to points, lines, or polygons. This process serves as the basis for the vectorization of images, in which a raster is converted into a vector dataset. An example of feature extraction is extracting a coastline from a satellite image and saving it as a vector dataset.

If this extraction is performed over several years, you could monitor the erosion or other changes along this coastline.

## Supervised and unsupervised classification

Objects on the Earth reflect different wavelengths of light, depending on the materials that they are made of. In remote sensing, analysts collect wavelength signatures for specific types of land cover (for example, concrete) and build a library for a specific area. A computer can then use this library to automatically locate classes in the library in a new image of the same area.

In unsupervised classification, a computer groups pixels with similar reflectance values in an image without any other reference information other than the histogram of the image.

## Creating the simplest possible Python GIS

Now that we have a better understanding of geospatial analysis, the next step is to build a simple GIS, known as **SimpleGIS**, using Python. This small program will be a technically complete GIS with a geographic data model that can render the data as a visual thematic map showing the population of different cities.

The data model will also be structured so that you can perform basic queries. Our SimpleGIS will contain the state of Colorado, three cities, and population counts for each city.

Most importantly, we will demonstrate the power and simplicity of Python programming by building this tiny system in pure Python. We will only use modules available in the standard Python distribution without downloading any third-party libraries.

## Getting started with Python

As we stated earlier, this book assumes that you have some basic knowledge of Python. The only module that's used in the following example is the `turtle` module, which provides a very simple graphics engine based on the `Tkinter` library, which is included with Python. In this book, we will use the Anaconda environment, which provides a Python environment plus a simple package manager called `conda` that will make installing any additional geospatial libraries we need easier. You can find the installer and installation instructions for your platform at <https://conda.io/projects/conda/en/latest/user-guide/install/index.html>. Once you have installed Anaconda, open the Anaconda PowerShell prompt. This will give you a command line to run Python interactively, as well as scripts.

Make sure that you can import the `turtle` module by typing the following in the command prompt. This will run the Turtle demo script:

```
python -m turtle
```

The preceding command will begin a real-time drawing program that will demonstrate the capabilities of the `turtle` module, as shown in the following screenshot:

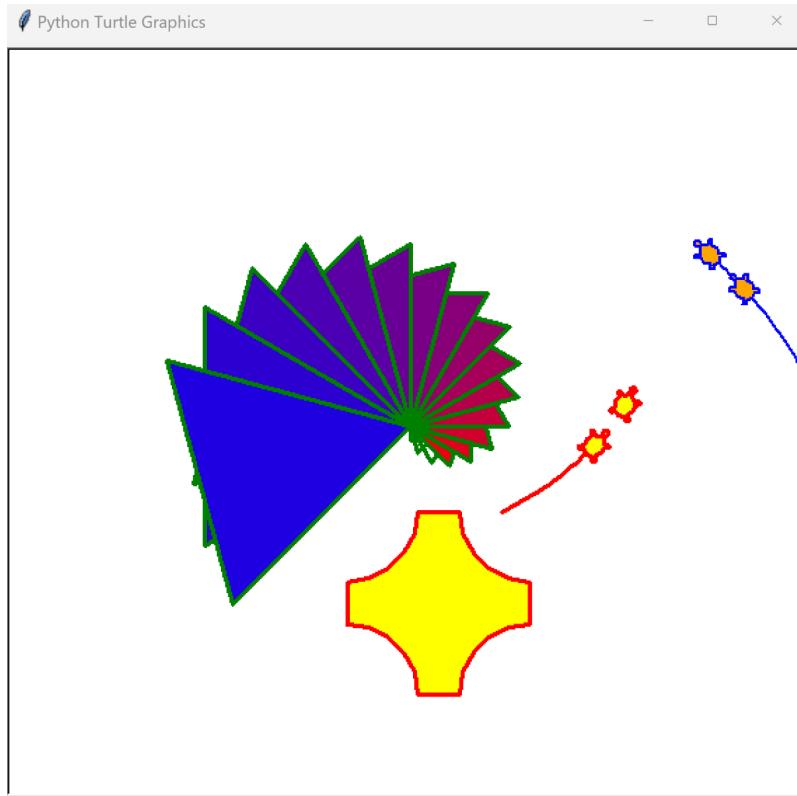


Figure 1.24 – The output of the Python Turtle graphics library

Now that we've seen what the `turtle` graphics module can do, let's use it to build an actual GIS!

## Building a SimpleGIS

The code is divided into two different sections:

- The data model section
- The map renderer that draws the data

For the data model, we will use simple Python lists. A Python list is a native data type that serves as a container for other Python objects in a specified order. Python lists can contain other lists and are great for simple data structures. They also map well to more complex structures or even databases if you decide you want to develop your script further.

The second portion of the code will render the map using the Python Turtle graphics engine. We will have only one function in the GIS that converts the world coordinates – in this case, longitude and latitude – into pixel coordinates. All graphics engines have an origin point of (0,0) and it's usually in the top-left or lower-left corner of the canvas.

Turtle graphics are designed to teach programming visually. The Turtle graphics canvas uses an origin of (0,0) in the center, similar to a graphing calculator. The following graph illustrates the type of Cartesian graph that the `turtle` module uses. Some of the points are plotted in both positive and negative space:

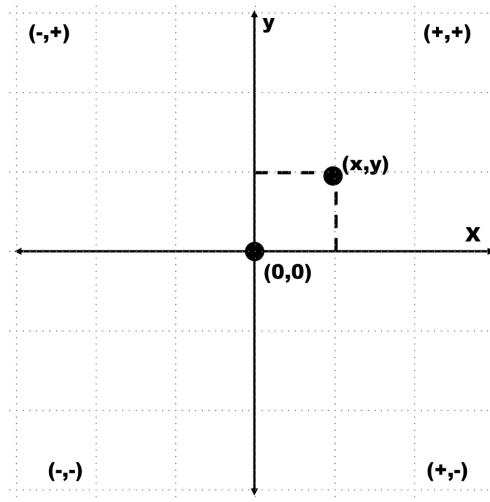


Figure 1.25 – The turtle module uses a standard Cartesian graph to plot coordinates

This also means that the Turtle graphics engine can have negative pixel coordinates, which is uncommon for graphics canvases. However, for this example, the `turtle` module is the quickest and simplest way for us to render our map.

### ***Setting up the data model***

You can run this program interactively in the Python interpreter or you can save the complete program as a script and run it. The Python interpreter is an incredibly powerful way to learn about new concepts because it gives you real-time feedback on errors or unexpected program behavior. You can easily recover from these issues and try something else until you get the results that you want:

1. In Python, you usually import modules at the beginning of the script, so we'll import the `turtle` module first. We'll use Python's import feature to assign the module the name `t` to save space and time when typing `turtle` commands:

```
import turtle as t
```

2. Next, we'll set up the data model, starting with some simple variables that allow us to access list indexes by name instead of numbers to make the code easier to follow. Python lists index the contained objects, starting with the number 0. So, if we wanted to access the first item in a list called `myList`, we would reference it as follows:

```
myList[0]
```

3. To make our code easier to read, we can also use a variable name that's been assigned to commonly used indexes:

```
firstItem = 0 myList[firstItem]
```

In computer science, assigning commonly used numbers to an easy-to-remember variable is a common practice. These variables are called constants. So, for our example, we'll assign constants for some common elements that are used for all of the cities. All cities will have a name, one or more points, and a population count:

```
NAME = 0  
POINTS = 1  
POP = 2
```

4. Now, we'll set up the data for Colorado as a list with a name, polygon points, and the population. Note that the coordinates are a list within a list:

```
state = ["COLORADO", [[-109, 37], [-109, 41], [-102, 41], [-102, 37]], 5187582]
```

5. The cities will be stored as nested lists. Each city's location consists of a single point as a longitude and latitude pair. These entries will complete our GIS data model. We'll start with an empty list called `cities` and then append the data to this list for each city:

```
cities = []  
cities.append(["DENVER", [-104.98, 39.74], 634265])  
cities.append(["BOULDER", [-105.27, 40.02], 98889])  
cities.append(["DURANGO", [-107.88, 37.28], 17069])
```

6. We will now render our GIS data as a map by first defining a map size. The width and height can be anything that you want, depending on your screen resolution:

```
map_width = 400  
map_height = 300
```

7. To scale the map to the graphics canvas, we must first determine the bounding box of the largest layer, which is the state. We'll set the map's bounding box to a global scale and reduce it to the size of the state. To do so, we'll loop through the longitude and latitude of each point and compare it with the current minimum and maximum X and Y values. If it is larger than the current maximum or smaller than the current minimum, we'll make this value the new maximum or minimum, respectively:

```
Minx =      180
Maxx =     -180
Miny =       90
Maxy =      -90
for x,y in state[POINTS]:
    if x < minx:
        minx = x
    elif x > maxx:
        maxx = x
    if y < miny:
        miny = y
    elif y > maxy:
        maxy = y
```

8. The second step when it comes to scaling is calculating a ratio between the actual state and the tiny canvas that we will render it on. This ratio is used for coordinate-to-pixel conversion. We get the size along the X and Y axes of the state and then we divide the map's width and height by these numbers to get our scaling ratio:

```
dist_x = maxx - minx
dist_y = maxy - miny
x_ratio = map_width / dist_x
y_ratio = map_height / dist_y
```

9. The following function, called `convert()`, is our only function in SimpleGIS. It transforms a point in the map coordinates from one of our data layers into pixel coordinates using the previous calculations. You'll notice that, in the end, we divide the map's width and height in half and subtract it from the final conversion to account for the unusual center origin of the Turtle graphics canvas. Every geospatial program has some form of this function:

```
def convert(point):
    lon = point[0] lat = point[1]
    x = map_width - ((maxx - lon) * x_ratio)
    y = map_height - ((maxy - lat) * y_ratio)
    # Python turtle graphics start in the
    # middle of the screen so we must offset
    # the points so they are centered
    x = x - (map_width/2)
    y = y - (map_height/2) return [x,y]
```

Now comes the exciting part! We're ready to render our GIS as a thematic map.

### ***Rendering the map***

The `turtle` module uses the concept of a cursor, known as a pen. Moving the cursor around the canvas is the same as moving a pen around a piece of paper. The cursor will draw a line when you move it. You'll notice that, throughout the code, we use the `t.up()` and `t.down()` commands to pick the pen up when we want to move to a new location and put it down when we're ready to draw. We have some important steps to follow in this section, so let's get started:

1. First, we need to set up the Turtle graphics window:

```
t.setup(900, 600)
t.screensize(800, 500)
wn = t.Screen()
wn.title("Simple GIS")
```

2. Since the border of Colorado is a polygon, we must draw a line between the last point and the first point to close the polygon. We can also leave out the closing step and just add a duplicate point to the Colorado dataset. Once we've drawn the state, we can use the `write()` method to label the polygon:

```
t.up()
first_pixel = None
for point in state[POINTS]:
    pixel = convert(point)
    if not first_pixel:
        first_pixel = pixel
    t.goto(pixel)
    t.down()
t.goto(first_pixel)
t.up() t.goto([0,0])
t.write(state[NAME], align="center",
\ font=("Arial",16,"bold"))
```

3. If we were to run the code at this point, we would see a simplified map of the state of Colorado, as shown in the following screenshot. Note that if you run the previous code, you need to temporarily add `t.done()` to the end to keep the window from automatically closing:

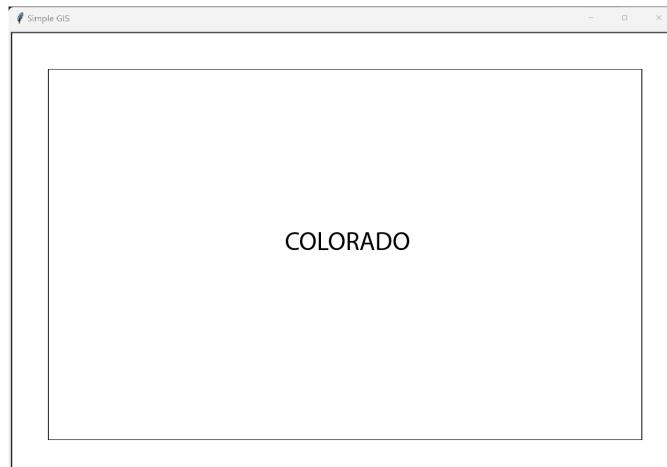


Figure 1.26 – A basemap of the state of Colorado, USA, produced by the SimpleGIS Python script

4. Now, we'll render the cities as point locations and label them with their names and population. Since the cities are a group of features in a list, we'll loop through them to render them. Instead of drawing lines by moving the pen around, we'll use Turtle's `dot()` method to plot a small circle at the pixel coordinate that's returned by our `SimpleGISconvert()` function. We'll then label the dot with the city's name and add the population. You'll notice that we must convert the population number into a string to use it in Turtle's `write()` method. To do so, we will use Python's built-in `str()` function:

```
for city in cities:  
    pixel = convert(city[POINTS]) t.up()  
    t.goto(pixel)  
    # Place a point for the city t.dot(10)  
    # Label the city  
    t.write(city[NAME] + " Pop.: " + str(city[POP])\n        ,  
            align="left")  
    t.up()
```

5. Now, we will perform one last operation to prove that we have created a real GIS. We will perform an attribute query on our data to determine which city has the largest population. Then, we'll perform a spatial query to see which city lies the furthest west. Finally, we'll print the answers to our questions on our thematic map page safely, out of the range of the map.

6. For our query engine, we'll use Python's built-in `min()` and `max()` functions. These functions take a list as an argument and return the minimum and maximum values of this list, respectively. These functions have a special feature called a key argument that allows you to sort complex objects. Since we are dealing with nested lists in our data model, we'll take advantage of the key argument in these functions. The key argument accepts a function that temporarily alters the list for evaluation before a final value is returned. In this case, we want to isolate the population values for comparison, and then the points. We could write a whole new function to return the specified value, but we can use Python's `lambda` keyword instead. The `lambda` keyword defines an anonymous function that is used inline. Other Python functions can be used inline, such as the string function, `str()`, but they are not anonymous. This temporary function will isolate our value of interest.
7. So, our first question is, which city has the largest population?

```
biggest_city = max(cities, key=lambda city:city[POP]) t.goto(0,-200)
t.write("The biggest city is: " + biggest_city[NAME])
```

8. The next question is, which city lies the furthest west?

```
western_city = min(cities, key=lambda city:city[POINTS])
t.goto(0,-220)
t.write("The western-most city is: " + western_city[NAME])
```

9. In the preceding query, we used Python's built-in `min()` function to select the smallest longitude value. This works because we represented our city locations as longitude and latitude pairs. It is possible to use different representations for points, including possible representations where this code would need to be modified to work correctly. However, for our SimpleGIS, we are using a common point representation to make it as intuitive as possible.
10. These last two commands are just for cleanup purposes. First, we hide the cursor. Then, we call Turtle's `done()` method, which will keep the Turtle graphics window with our map on it open until we choose to close it using the close handle at the top of the window:

```
t.pen(shown=False)
t.done()
```

Whether you followed along using the Python interpreter or you ran the complete program as a script, you should see the following map being rendered in real time:

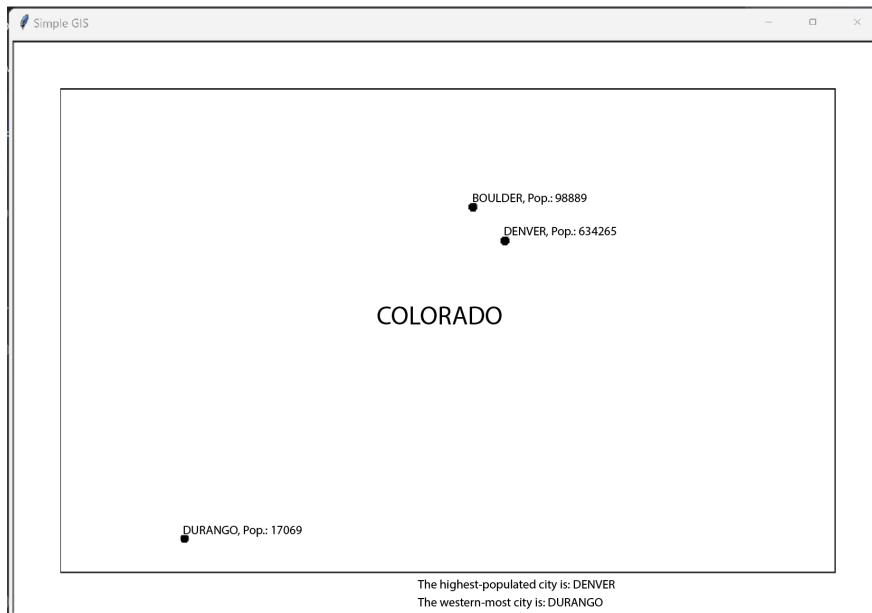


Figure 1.27 – The final output of the SimpleGIS script illustrating a complete GIS in only 60 lines of Python

Congratulations! You have followed in the footsteps of Paleolithic hunters, the father of GIS Dr. Roger Tomlinson, geospatial pioneer Howard Fisher, and game-changing humanitarian programmers to create a functional, extensible, and technically complete GIS.

It took less than 60 lines of pure Python code! You will be hard-pressed to find a programming language that can create a complete GIS using only its core libraries in such a finite amount of readable code as Python. Even if you did, it is highly unlikely that the language would survive the geospatial Python journey that you'll take through the rest of this book.

As you can see, there is lots of room for expansion when it comes to SimpleGIS. Here are some other ways that you might expand this simple tool using the reference material for Tkinter and Python that were linked at the beginning of this section:

- Create an overview map in the top-right corner with a US border outline and Colorado's location in the US
- Add color for visual appeal and further clarity
- Create a map key for different features
- Make a list of states and cities and add more states and cities
- Add a title to the map
- Create a bar chart to compare population numbers visually

The possibilities are endless. SimpleGIS can also be used as a way to quickly test and visualize geospatial algorithms that you come across. If you want to add more data layers, you can create more lists, but these lists will become difficult to manage. In this case, you can use another Python module that's included in the standard distribution. The SQLite module provides a SQL-like database in Python that can be saved to disk or run in memory.

## Summary

Well done! You are now a geospatial analyst. In this chapter, you learned about the history of geospatial analysis and the technologies that support it. You saw how geospatial technology led the fight against the COVID-19 pandemic. You also became familiar with foundational GIS and remote sensing concepts that will serve you through the rest of this book. Finally, you took all of this knowledge and built a working GIS that can be expanded to do whatever you can imagine!

In the next chapter, we'll tackle the data formats that you'll encounter as a geospatial analyst. Geospatial analysts spend far more time dealing with data than actually performing analysis. Understanding the data that you're working with is essential to working efficiently and having fun.

## Questions

1. Which government agency first coined the term GIS?
2. What is a map called that is based on a theme?
3. What are the two core capabilities that constitute a GIS?

## Further reading

- <https://gisgeography.com/history-of-gis/>
- <https://education.nationalgeographic.org/resource/geographic-information-system-gis/>



# 2

## Learning about Geospatial Data

One of the most challenging aspects of geospatial analysis is the data. Geospatial data already includes dozens of file formats and database structures and continues to evolve and grow to include new types of data and standards. Additionally, almost any file format can technically contain geospatial information by simply adding a location.

In this chapter, we will look at the following topics:

- Overview of common data formats
- Understanding data structures
- Understanding spatial indexing
- What are overviews?
- What is metadata?
- Understanding the file structure
- Knowing about the most widely used vector data types
- Understanding raster data types
- What is point cloud data?
- More realistic geospatial models with 3D data
- What are web services?
- Understanding geospatial databases
- Sharing data with interchangeable formats
- Introducing spatiotemporal data

By the end of this chapter, you'll be familiar with most geospatial data formats and what they are used for in geospatial analysis.

## Technical requirements

You can find the code files for this chapter on GitHub at [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730\\_02\\_Asset\\_Files](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730_02_Asset_Files).

## Overview of common data formats

As a geospatial professional, you may frequently encounter the following general data types:

Spreadsheets and **comma-separated values (CSV)** files or **tab-separated values (TSV)** files:

- Geotagged photos
- Lightweight binary points, lines, and polygons
- Multi-gigabyte satellite or aerial images
- Elevation data such as grids, point clouds, or integer-based images
- XML files
- JSON files
- Databases (both servers and file databases)
- Web services
- Geodatabases

Each format contains its own challenges for access and processing. When you perform analysis on data, you usually have to do some form of preprocessing first. You might clip or subset a satellite image of a large area down to just your area of interest, or you might reduce the number of points in a collection to just the ones meeting certain criteria in your data model. A good example of this type of preprocessing is the SimpleGIS example that we looked at at the end of *Chapter 1, Learning about Geospatial Analysis with Python*. The state dataset included just the state of Colorado rather than all 50 states. The city dataset included only three sample cities, demonstrating three levels of population, along with different relative locations.

The common geospatial operations in *Chapter 1, Learning about Geospatial Analysis with Python*, are the building blocks for this type of preprocessing. However, it is important to note that there has been a gradual shift in the field of geospatial analysis toward readily available basemaps. Until around 2004, geospatial data was difficult to acquire and desktop computing was much less powerful than it is today. Preprocessing data was an absolute first step for any geospatial project. However, in 2005, Google released Google Maps, which wasn't long after Google Earth. Microsoft had also been developing a technology acquisition called **TerraServer**, which it relaunched around that time. In 2004, the **Open Geospatial Consortium (OGC)** updated the version of its **Web Map Service (WMS)** to version 1.3.0, which was growing in use and popularity. That same year, Esri also released version 9

of its ArcGIS server system. These innovations were driven by Google's web map tiling model, which allowed for smooth, global, scrolling maps at many different resolutions, often called slippy maps. Now this method of data storage and display is called **Tiled Maps**.

People used map servers on the internet before Google Maps, most famously with the MapQuest driving directions website. However, these map servers offered only small amounts of data at a time and usually over limited areas. The Google tiling system converted global maps into tiered image tiles for both images and mapping data. These were served dynamically using JavaScript and the browser-based XMLHttpRequest API, more commonly known as **Asynchronous JavaScript and XML (AJAX)**. Google's system scaled to millions of users using ordinary web browsers. More importantly, it allowed programmers to leverage JavaScript programming to create mashups so that they could use the Google Maps JavaScript API to add additional data to maps. The mashup concept is actually a shared geospatial layer system. Users can combine and recombine data from different web services into a single map, as long as the data is web-accessible. Other commercial and open source systems quickly mimicked this concept.

A notable example of distributed geospatial layers is OpenLayers, which provides an open source, Google-like API that has now gone beyond Google's API, offering additional features. Complementary to OpenLayers is OpenStreetMap, which is the open source answer to the tiled map services consumed by systems such as OpenLayers.

OpenStreetMap has global, street-level vector data and other spatial features that have been collected from available government data sources and the contributions of thousands of editors worldwide. OpenStreetMap's data maintenance model is similar to the way Wikipedia, the online encyclopedia, crowdsources information creation and updates for articles. Recently, even more mapping APIs have appeared, including Leaflet and Mapbox, which continue to increase in flexibility, simplicity, and capability.

The mashup revolution had interesting and beneficial side effects on data. Geospatial data is traditionally difficult to obtain. The cost of collecting, processing, and distributing data kept geospatial analysis constrained to those who could afford this steep overhead by producing data or purchasing it. For decades, geospatial analysis was the tool of governments, very large organizations, and universities. Once the web mapping trend shifted to large-scale, globally tiled maps, organizations began essentially providing basemap layers for free in order to draw developers to their platform. The massively scalable global map system required massively scalable, high-resolution data to be useful. Geospatial software producers and data providers wanted to maintain their market share and kept up with the technology trend.

Geospatial analysts benefited greatly from this market shift in several ways. First of all, data providers began distributing data in a common projection called **Mercator**. The Mercator projection is a nautical navigation projection that was introduced over 400 years ago. As we mentioned in *Chapter 1, Learning about Geospatial Analysis with Python*, all projections have practical benefits, as well as distortions. The distortion in the Mercator projection is its size. In a global view, Greenland appears bigger than the continent of South America. However, like every projection, it also has some benefits. Mercator

preserves angles. Predictable angles allowed medieval navigators to draw straight bearing lines when plotting a course across oceans. Google Maps didn't launch with Mercator. However, it quickly became clear that roads in high and low latitudes met at odd angles on the map instead of 90 degrees in reality.

Since the primary purpose of Google Maps was street-level driving directions, Google sacrificed global view accuracy for far better relative accuracy of streets when viewing a single city. Competing mapping systems followed suit. Google also standardized on the WGS 84 datum. This datum defines a specific spherical model of Earth, called a **geoid**. This model defines the normal sea level. What's significant about this choice by Google is that the **Global Positioning System (GPS)** also uses this datum. Therefore, most GPS units default to this datum as well, making Google Maps easily compatible with raw GIS data.

The Google variation of the Mercator projection is often called **Google Mercator**. The **European Petroleum Survey Group (EPSG)** assigns short numerical codes to projections as an easy way to reference them. Rather than waiting for the EPSG to approve or assign a code that was first only relevant to Google, they began calling the projection EPSG:900913, which is Google spelled with numbers. Later, EPSG assigned this project code EPSG:3857, deprecating the older code. Most GIS systems recognize the two codes as synonymous. It should be noted that Google tweaked the standard Mercator projection slightly for its use; however, this variation is almost imperceptible. Google uses spherical formulas at all map scales, while the standard Mercator assumes an ellipsoidal form at large scales.

The following image of the Mercator projection ([https://en.wikipedia.org/wiki/File:Tissot\\_mercator.png](https://en.wikipedia.org/wiki/File:Tissot_mercator.png)) was taken from Wikipedia:

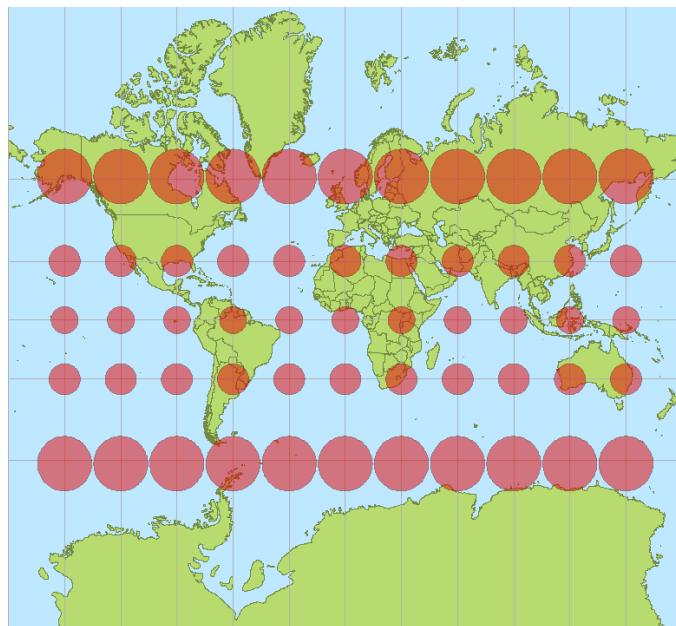


Figure 2.1 – Tissot's indicatrix showing map projection distortion

It shows the distortion caused by the Mercator projection using Tissot's indicatrix, which projects small ellipses of equal size on a map. The distortion of the ellipse clearly shows how the projection affects the size and distance: web mapping services have reduced the chore of hunting for data and much of the preprocessing for analysts to create basemaps. However, to create anything of value, you must understand geospatial data and how to work with it. This chapter provides an overview of the common data types and issues that you will encounter in geospatial analysis.

Throughout this chapter, two terms will be commonly used:

- **Vector data:** Vector data includes any format that minimally represents geolocation data using points, lines, or polygons.
- **Raster data:** Raster data includes any format that stores data in a grid of rows and columns. Raster data includes all image formats.

These are the two primary categories under which most geospatial datasets can be grouped.

#### More on map projections

If you want to see a projection that shows the relative size of continents more accurately, refer to the Goode homolosine projection at [https://en.wikipedia.org/wiki/Goode\\_homolosine\\_projection](https://en.wikipedia.org/wiki/Goode_homolosine_projection).

Now that we understand the high-level differences of different geospatial data types, let's look at what they have in common.

## Understanding data structures

Despite having dozens of formats, geospatial data has some common traits. Understanding these traits can help you approach and understand unfamiliar data formats by identifying the ingredients common to nearly all spatial data. The structure of a given data format is usually driven by its intended use.

Some data is optimized for efficient storage or compression, some is optimized for efficient access, and some is designed to be lightweight and readable (web formats), while other data formats seek to contain as many different data types as possible.

Interestingly, some of the most popular formats today are also some of the simplest and even lack features found in more capable and sophisticated formats. Ease of use is extremely important to geospatial analysts because so much time is spent integrating data into geographic information systems, as well as exchanging data among analysts. Simple data formats facilitate these activities the best.

## Common traits

Geospatial analysis is an approach in which you apply information processing techniques to data with a geographic context. This definition contains the most important elements of geospatial data:

- **Geolocation data:** Geolocation information can be as simple as a single point on Earth referencing where a photo was taken. It can also be as complex as a satellite camera engineering model and orbital mechanics information being used to reconstruct the exact conditions and location under which the satellite captured the image.
- **Subject information:** Subject information can also cover a wide range of possibilities. Sometimes, the pixels in an image are the data in terms of a visual representation of the ground. Other times, an image may be processed using multispectral bands, such as infrared light, to provide information that's not visible in the image. Processed images are often classified using a structured color palette that is linked to a key, describing the information each color represents. Other possibilities include some form of database with rows and columns of information for each geolocated feature, such as the population associated with each city in our SimpleGIS program from *Chapter 1, Learning about Geospatial Analysis with Python*.

These two factors are present in every format that can be considered geospatial data. Another common feature of geospatial data is spatial indexing. Overview datasets are also related to indexing.

## Understanding spatial indexing

Geospatial datasets are often very large files, easily reaching hundreds of megabytes or even several gigabytes in size. Geospatial software can be quite slow in trying to repeatedly access large files when performing analysis.

As discussed briefly in *Chapter 1, Learning about Geospatial Analysis with Python*, spatial indexing creates a guide, which allows the software to quickly locate query results without examining every single feature in the dataset. Spatial indexes allow the software to eliminate possibilities and perform more detailed searches or comparisons on a much smaller subset of the data.

## Spatial indexing algorithms

Many spatial indexing algorithms are derivatives of well-established algorithms that have been used on non-spatial information for decades. The two most common spatial indexing algorithms are **quadtree index** and **R-tree index**. There is a three-dimensional counterpart to the Quadtree index called an **octree data structure**. This indexing structure is most commonly seen with point cloud data – specifically, Lidar data.

## Quadtrees index

The quadtree algorithm actually represents a series of different algorithms based on a common theme. Each node in a quadtree index contains four children. These child nodes are typically square or rectangular in shape. When a node contains a specified number of features and more features are added, the node splits.

The concept of dividing a space into nested squares speeds up spatial searches. The software must only handle five points at a time and use simple greater-than/less-than comparisons to check whether a point is inside a node. Quadtree indexes are most commonly found in file-based index formats.

The following diagram shows a point dataset sorted by a quadtree algorithm. The black points are the actual dataset, while the boxes are the bounding boxes of the index. Note that none of the bounding boxes overlap. The diagram on the left shows the spatial representation of the index, while the diagram on the right shows the hierarchical relationship of a typical index, which is how spatial software sees the index and data.

This structure allows a spatial search algorithm to quickly eliminate possibilities when trying to locate one or more points in relation to some other set of features, as shown in the following diagram:

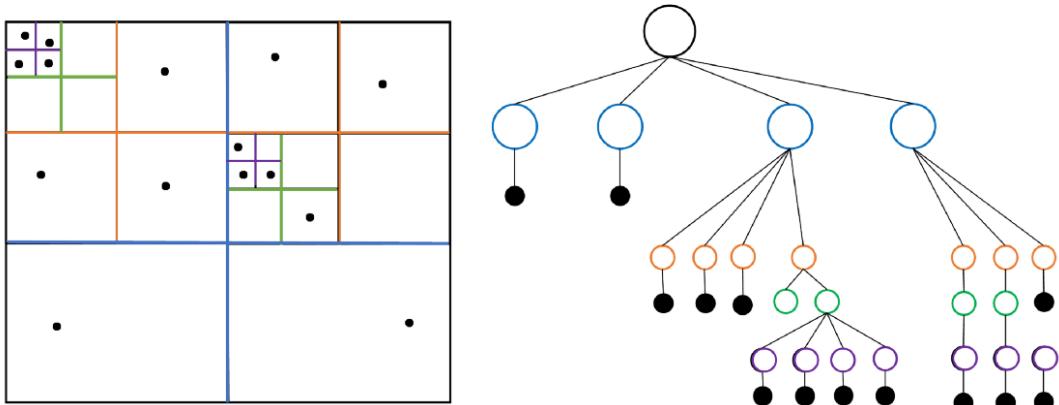


Figure 2.2 – Visual representation of a quadtree index spatially and organizationally

Now that we understand quadtree indexes, let's look at another common type of spatial indexes, called R-trees.

## R-tree indexes

**R-tree** indexes are more sophisticated than quadtrees. R-trees are designed to handle 3D data and are optimized to store the index in a way that is compatible with the way databases use disk space and memory. Nearby objects are grouped together using an algorithm from a variety of spatial algorithms. All objects in a group are bounded by a minimum rectangle. These rectangles are aggregated into hierarchical nodes that are balanced at each level.

Unlike a quadtree, the bounding boxes of an R-tree may overlap across nodes. Due to their relative complexity and database-oriented structure, R-trees are most commonly found in spatial databases, as opposed to file-based formats.

The following diagram from <https://en.wikipedia.org/wiki/File:R-tree.svg> shows a balanced R-tree for a 2D point dataset:

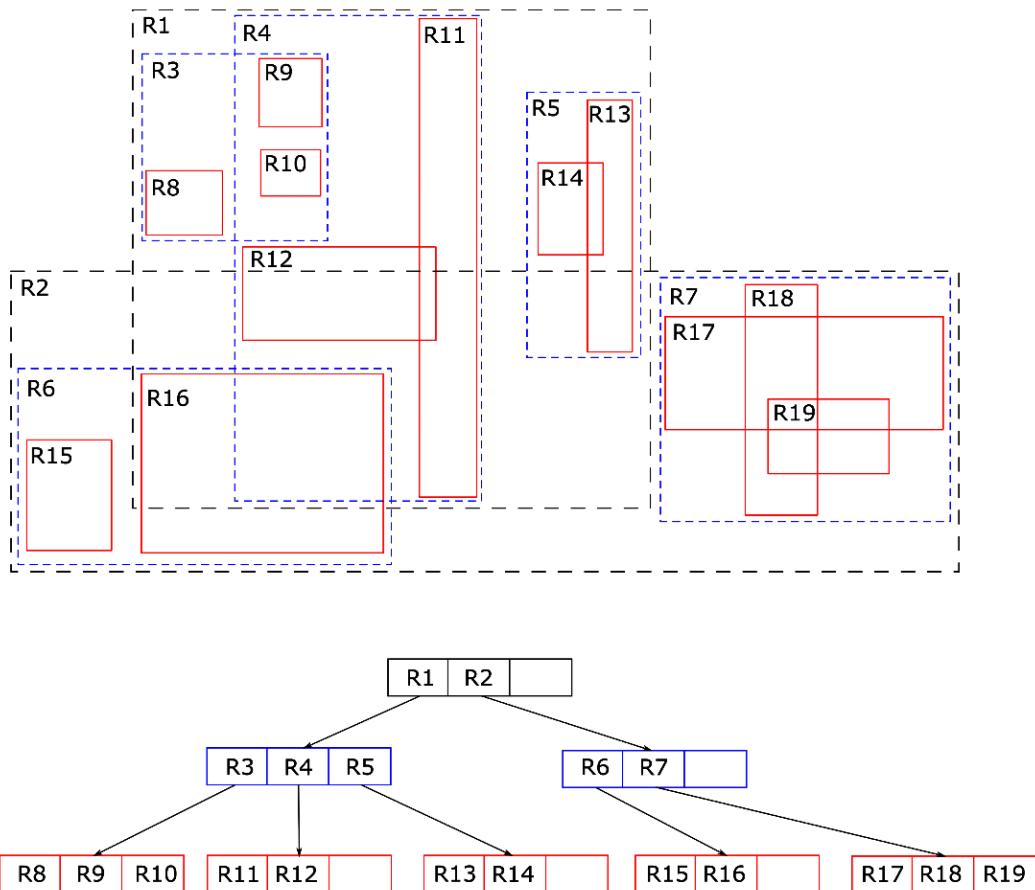


Figure 2.3 – Visual representation of an R-tree index spatially and organizationally

Indexes break up large datasets, but to speed up searching, they may employ a technique called grids. We'll look at that next.

## Grids

Spatial indexes also often employ the concept of an integer grid. Geospatial coordinates are usually floating-point decimal numbers with anywhere from 2 to 16 decimal places.

Performing comparisons on floating-point numbers is far more computationally expensive than working with integers. Indexed searching is about eliminating possibilities that don't require precision first.

Most spatial indexing algorithms, therefore, map floating-point coordinates to a fixed-sized integer grid. On searching for a particular feature, the software can use more efficient integer comparisons rather than working with floating-point numbers. Once the results are narrowed down, the software can access the full resolution data.

Grid sizes can be as small as 256 x 256 for simple file formats, or can be as large as 3 million x 3 million in large geospatial databases designed to incorporate every known coordinate system and possible resolution.

The integer mapping technique is very similar to the rendering technique that is used to plot data on a graphics canvas in mapping programs. The SimpleGIS script in *Chapter 1, Learning about Geospatial Analysis with Python*, also uses this technique to render points and polygons using the built-in Python turtle graphics engine.

Spatial indexing is used to speed up the searching and displaying of vector data. There's a different technique to speed up the rendering of raster data. Let's take a look at that method.

## What are overviews?

Overview data is most commonly found in raster formats. Overviews are resampled and lower-resolution versions of raster datasets that provide thumbnail views or simply faster-loading image views at different map scales. They are also known as pyramids, and the process of creating them is known as pyramiding an image. These overviews are usually preprocessed and stored with full resolution data either embedded in the file or in a separate file.

The compromise of this convenience is that the additional images add to the overall file size of the dataset; however, they speed up image viewers. Vector data also has a concept of overviews, usually to give a dataset geographic context in an overview map. However, because vector data is scalable, reduced-size overviews are usually created on the fly by software using a generalization operation, as mentioned in *Chapter 1, Learning about Geospatial Analysis with Python*.

Occasionally, vector data is rasterized by converting it into a thumbnail image, which is stored with, or embedded in, the image header. The following diagram demonstrates the concept of image overviews and shows why they are often called pyramids:

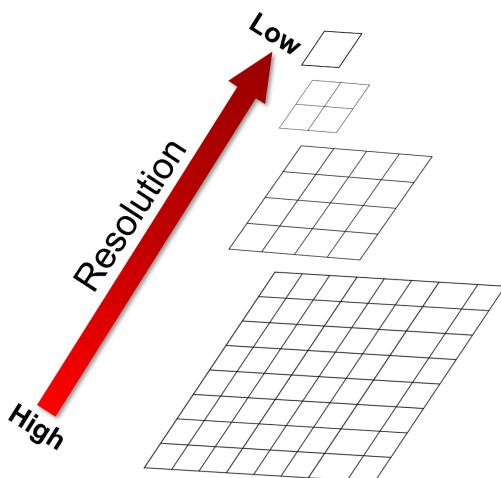


Figure 2.4 – Resolution scaling of raster image pyramids

Spatial indexing and overviews help speed up access to data by software for analysts. Next, we'll look at metadata, which provides both a human- and machine-readable way to understand, search, and even catalog data.

## What is metadata?

As discussed in *Chapter 1, Learning about Geospatial Analysis with Python*, metadata is any data that describes the associated dataset. Common examples of metadata include basic elements such as the footprint of the dataset on Earth, as well as more detailed information such as spatial projection and information describing how the dataset was created.

Most data formats contain the footprint or bounding box of the data on Earth. Detailed metadata is typically stored in a separate location in a standard format, such as the **US Federal Geographic Data Committee (FGDC)**, **Content Standard for Digital Geospatial Metadata (CSDGM)**, ISO, or the newer European Union initiative, which includes metadata requirements, and is called the **Infrastructure for Spatial Information in the European Community (INSPIRE)**.

## Understanding the file structure

The preceding elements can be stored in a variety of ways in a single file, multiple files, or databases, depending on the format. Additionally, this geospatial information can be stored in a variety of formats, including embedded binary headers, XML, database tables, spreadsheets/CSV, separate text, or binary files.

Human-readable formats such as XML files, spreadsheets, and structured text files require only a text editor to be investigated. These files are also easily parsed and processed using Python's built-in

---

modules, data types, and string manipulation functions. Binary-based formats are more complicated. Therefore, it is typically easier to use a third-party library to deal with binary formats.

However, you don't have to use a third-party library, especially if you just want to investigate the data at a high level. Python's built-in `struct` module has everything that you need. The `struct` module lets you read and write binary data as strings. When using the `struct` module, you need to be aware of the concept of byte order. Byte order refers to how the bytes of information that make up a file are stored in memory. This order is usually platform-specific, but in some rare cases, including shapefiles, the byte order is mixed into the file.

The Python `struct` module uses the greater than (`>`) and less than (`<`) symbols to specify byte order (big-endian and little-endian, respectively).

The following brief example demonstrates the usage of the Python `struct` module to parse the bounding box coordinates from an Esri shapefile vector dataset. You can download this shapefile as a zipped file from the following URL: [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_02\\_Asset\\_Files/hancock.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_02_Asset_Files/hancock.zip).

When you unzip this, you will see three files. For this example, we'll be using `hancock.shp`. The Esri shapefile format has a fixed location and data type in the file header from byte 36 to byte 37 for the minimum *x*, minimum *y*, maximum *x*, and maximum *y* bounding box values. In this example, we will execute the following steps:

1. Import the `struct` module.
2. Open the `hancock.zip` shapefile in the binary read mode.
3. Navigate to byte 36.
4. Read each 8-byte double variable specified as `d`, and unpack it using the `struct` module in little-endian order, as designated by the `<` sign.

The best way to execute this script is in the interactive Python interpreter. We will then read the minimum longitude, minimum latitude, maximum longitude, and maximum latitude:

```
>>> import struct
>>> f = open("hancock.shp", "rb")
>>> f.seek(36)
>>> struct.unpack("<d", f.read(8))
(-89.6904544701547,)
>>> struct.unpack("<d", f.read(8))
(30.173943486533133,)
>>> struct.unpack("<d", f.read(8))
(-89.32227546981174,)
>>> struct.unpack("<d", f.read(8))
(30.6483914869749,)
```

You'll notice that, when the `struct` module unpacks a value, it returns a Python tuple with one value. You can shorten the preceding unpacking code to one line by specifying all four doubles at once and increasing the byte length to 32 bytes, as shown in the following code:

```
>>> f.seek(36)
>>> struct.unpack("<dddd", f.read(32))
(-89.6904544701547, 30.173943486533133, -89.32227546981174,
 30.6483914869749)
```

Now that we understand how to describe data, let's learn about one of the most common types of geospatial data—**vector data**.

## Knowing about the most widely used vector data types

Vector data is, by far, the most common geospatial format because it is the most efficient way to store spatial information. In general, it requires fewer computer resources to store and process than raster data. The OGC has over 16 formats directly related to vector data. Vector data stores only geometric primitives, including points, lines, and polygons.

However, only the points are stored for each type of shape. For example, in the case of a simple straight vector line shape, only the endpoints would be necessarily stored and defined as a line. Software displaying this data would read the shape type and then connect the endpoints with a line dynamically.

Geospatial vector data is similar to the concept of vector computer graphics, with some notable exceptions. Geospatial vector data contains positive and negative Earth-based coordinates, while vector graphics typically store computer screen coordinates. Geospatial vector data is also usually linked to other information about the object that's represented by the geometry. This information may be as simple as a timestamp in the case of GPS data or an entire database table for larger geographic information systems.

Vector graphics often store styling information describing colors, shadows, and other display-related instructions, while geospatial vector data typically does not. Another important difference is the shapes. Geospatial vectors typically only include very primitive geometries based on points, straight lines, and straight-line polygons, while many computer graphic vector formats have concepts of curves and circles. However, geospatial vectors can model these shapes using more points.

Other human-readable formats, such as CSV, simple text strings, GeoJSON, and XML-based formats, are technically vector data because they store geometry as opposed to rasters, which represent all the data within the bounding box of the dataset. Until the explosion of XML in the late 1990s, vector data formats were nearly all binary. XML provided a hybrid approach that was both computer- and human-readable. The compromise is that text formats such as GeoJSON and XML data greatly increase the file size compared to binary formats. These formats will be discussed later in this section.

The number of vector formats to choose from is staggering. The open source vector library, **OGR** (<https://gdal.org/drivers/vector/index.html>), lists over 86 supported vector formats. Its commercial counterpart, Safe Software's **Feature Manipulation Engine (FME)**, lists over 188 supported vector formats (<https://www.safe.com/fme/formats-matrix/#platforms>). These lists include a few vector graphics formats, as well as human-readable geospatial formats. There are still dozens of formats out there to at least be aware of, in case you come across them.

Now, let's look at a specific and widely used type of vector data called shapefiles.

## Shapefiles

The most ubiquitous geospatial format is the Esri shapefile. The geospatial software company known as Esri released the shapefile format specification as an open format in 1998 (<http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf>). Esri developed it as a format for their ArcView software, designed as a lower-end GIS option to complement their high-end professional package, ArcInfo, formerly called ARC/INFO. However, the open specification, efficiency, and simplicity of the format turned it into an unofficial GIS standard that is still extremely popular over 25 years later.

Virtually every piece of software labeled as geospatial software supports shapefiles because the shapefile format is so common. For this reason, you can almost get by as an analyst by being intimately familiar with shapefiles and mostly ignoring other formats. You can convert almost any other format into a shapefile through the source format's native software or a third-party converter, such as the OGR library, for which there is a Python module. Other Python modules that handle shapefiles are PyShp, Fiona, and Pandas, with the latter two based on OGR.

One of the most striking features of a shapefile is that the format consists of multiple files (from minimum to maximum, there can be 3-15 different files). The following table describes the file formats. The .shp, .shx, and .dbf files are required for a valid shapefile.

Shapefile supporting file extension	Supporting file purpose	Notes
.shp	This is the shapefile. It contains the geometry.	It is a required file. Some software that needs only geometry will accept .shp files without the .shx or .dbf files.
.shx	This is the shape index file. It is a fixed-sized record index referencing geometry for faster access.	It is a required file. This file is meaningless without the .shp file.

.dbf	This is the database file. It contains the geometry attributes.	It is a required file. Some software will access this format without the .shp file present as the specification predates shapefiles. It's based on the very old FoxPro and dBase formats. An open specification exists for it called Xbase. The .dbf files can be opened by most types of spreadsheet software.
.sbn	This is the spatial bin file, that is, the shapefile spatial index.	It contains bounding boxes of features mapped to a 256 x 256 integer grid. It is very common for this file to accompany large shapefile datasets.
.sbx	A fixed-sized record index for the .sbn file.	A traditional ordered record index of a spatial index. Commonly found in most datasets along with the .sbn file to speed up spatial queries.
.prj	This contains map projection information that's stored in a well-known text format.	A very common and required file for on-the-fly projection by the GIS software. This same format can also accompany raster data. Though it is not mandatory to include it, it is an established best practice to include this file when sharing data as it is essential to using the data in a modern GIS.
.fbn	A spatial index of read-only features.	Very rarely seen.
.fbx	A fixed-sized record index of the .fbn spatial index.	Very rarely seen.
.ixs	A geocoding index.	Common in geocoding applications, including driving-direction type applications.
.mxs	Another type of geocoding index.	Less common than the .ixs format.
.ain	Attribute index.	Mostly legacy format, and rarely used in modern software.
.aih	Attribute index.	Accompanies the .ain files.
.qix	Quadtree index.	A spatial index format created by the open source community because Esri .sbn and .sbx files were undocumented until recently.
.atx	Attribute index.	A more recent Esri software-specific attribute index to speed up attribute queries.

.shp.xml	Metadata.	A geospatial metadata .xml container. It can be any of the multiple XML standards, including FGDC and ISO.
.cpg	Code page file for .dbf.	It is used for the internationalization of .dbf files to identify the character set the shapefile uses.

Table 2.1 – The 16 different files possible within a single shapefile dataset

You will probably never encounter all of these formats at once. However, any shapefile that you use will have multiple files. You will commonly see .shp, .shx, .dbf, .prj, .sbn, .sbx, and occasionally, .shp.xml files. If you want to rename a shapefile, you must rename all of the associated files with the same name; however, in Esri software and other GIS packages, these datasets will appear as a single file.

Another important feature of shapefiles is that the records are not numbered. Records include the geometry, the .shx index record, and the .dbf record. These records are stored in a fixed order. When you examine shapefile records using the software, they appear to be numbered.

People are often confused when they delete a shapefile record, save the file, and reopen it; the number of the record that was deleted still appears. The reason for this is that shapefile records are numbered dynamically on loading, but not saved. So, for example, if you delete record number 23 and save the shapefile, record number 24 will become 23 the next time you read the shapefile. Many people expect to open the shapefile and see the records jump from 22 to 24. The only way to track shapefile records in this way is to create a new attribute called ID or similar in the .dbf file and assign each record a permanent and unique identifier.

Just like renaming shapefiles, care must be taken while editing shapefiles. It's best to use software that treats the shapefiles as a single dataset. If you edit any of the files individually and add/delete a record without editing the accompanying files, the shapefile will be seen as corrupt by most geospatial software.

## CAD files

CAD stands for **computer-aided design**. The primary formats for CAD data were created by Autodesk for their leading AutoCAD package. The two formats that are commonly seen are **Drawing Exchange Format (DXF)** and AutoCAD's native **Drawing (DWG)** format.

CAD software is used for everything that is engineering-related, from designing bicycles to cars, parks, and city sewer systems. As a geospatial analyst, you don't have to worry about mechanical engineering designs; however, civil engineering designs become quite an issue. Most engineering firms use geospatial analysis to a very limited degree but store nearly all of their data in the CAD format. The DWG and DXF formats can represent objects using features not found in geospatial software or that are weakly supported by geospatial systems. Some examples of these features include the following:

- Curves
- Surfaces (for objects that are different from geospatial elevation surfaces)
- 3D solids
- Text (rendered as an object)
- Text styling
- Viewport configuration

These CAD and engineering-specific features make it difficult to cleanly convert CAD data into geospatial formats. If you encounter CAD data, the easiest option is to ask the data provider whether they have shapefiles or some other geospatial-centric format.

## Tag-based and markup-based formats

Tag-based markup formats are typically XML formats. They also include other structured text formats such as the **Well-Known Text (WKT)** format, which is used for projection information files as well as different types of data exchange.

XML formats include the **Keyhole Markup Language (KML)**, the **OpenStreetMap (OSM)** format, and the Garmin GPX format for GPS data, which has become a popular exchange format. The Open Geospatial Consortium's **Geographic Markup Language (GML)** standard is one of the oldest and most widely used XML-based geographic formats. It is also the basis for the OGC **Web Feature Service (WFS)** standard for web applications.

However, GML has been largely superseded by KML and the GeoJSON format.

XML formats often contain more than just geometry. They also contain attributes and rendering instructions such as color, styling, and symbology. Google's KML format has become a fully supported OGC standard. The following is a sample of KML showing a simple placemark that is a point location on Earth with information about what that point is, which in this case is a coffee shop:

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Placemark>
    <name>Mockingbird Cafe</name>
```

```
<description>Coffee Shop</description>
<Point>
    <coordinates>-89.329160,30.310964</coordinates>
</Point>
</Placemark>
</kml>
```

The XML format is attractive to geospatial analysts for the following reasons:

- It is a human-readable format.
- It can be edited in a text editor.
- It is well supported by programming languages (especially Python). It is, by definition, easily extensible.

XML is not perfect, though. It is an inefficient storage mechanism for very large data formats and can quickly become cumbersome to edit. Errors in datasets are common, and most parsers do not handle errors robustly. Despite the downsides, XML is widely used in geospatial analysis.

**Scalable Vector Graphics (SVG)** is a widely supported XML format for computer graphics. It is supported well by browsers and is often used for geospatial rendering. However, SVG was not designed as a geographic format.

The WKT format is also an older OGC standard. The most common use for it is to define projection information usually stored in .prj projection files, along with a shapefile or raster. The WKT string for the WGS 84 coordinate system is as follows:

```
GEOGCS ["WGS 84",
    DATUM ["WGS_1984",
        SPHEROID ["WGS 84", 6378137, 298.257223563,
            AUTHORITY ["EPSG", "7030"] ],
        AUTHORITY ["EPSG", "6326"] ],
    PRIMEM ["Greenwich", 0,
        AUTHORITY ["EPSG", "8901"] ],
    UNIT ["degree", 0.01745329251994328,
        AUTHORITY ["EPSG", "9122"] ],
    AUTHORITY ["EPSG", "4326"] ]
```

The parameters that define a projection can be quite long. A standards committee, which was created by the EPSG, introduced a numerical coding system to reference projections. These codes, such as EPSG:4326, are used as shorthand for strings such as the preceding code. There are also short names for commonly used projections such as Mercator, which can be used in different software packages to reference a projection.

### More on reference systems

More information on these reference systems can be found on the Spatial Reference website at <http://spatialreference.org/ref/>.

## GeoJSON

GeoJSON is a brilliant text format based on the **JavaScript Object Notation (JSON)** format, which has been a commonly used data exchange format for years. Despite its short history, GeoJSON can be found embedded in all major geospatial software systems and most websites that distribute data. This is because JavaScript is the language of the dynamic web, and GeoJSON can be directly fed into JavaScript. It is also easily parsed in common programming languages in addition to JavaScript, such as C#, PHP, Java, and Python.

GeoJSON is a completely backward-compatible extension for the popular JSON format. The structure of JSON is very similar and, in some cases, identical to existing data structures of common programming languages. JSON is almost identical to Python's dictionary and list data types. Due to this similarity, parsing JSON in a script is simple to do from scratch, but there are many libraries to make it even easier. Python contains a built-in library aptly named `json`.

GeoJSON provides you with a standard way to define geometry, attributes, bounding boxes, and projection information. GeoJSON has all of the advantages of XML, including human-readable syntax, excellent software support, and wide use in the industry. It also surpasses XML.

GeoJSON is far more compact than XML, largely because it uses name/value pairs to define objects rather than opening and closing text-laden tags. This compactness also helps with the readability and manageability of larger datasets. However, it is still inferior to binary formats from a data volume standpoint. The following is a sample of the GeoJSON syntax, defining a geometry collection with both a point and line:

```
{ "type": "GeometryCollection",
  "geometries": [
    { "type": "Point",
      "coordinates": [
        [-89.33,30]
      ],
      "type": "LineString",
      "coordinates": [
        [-89.33,30.3], [-89.36,30.28]
      ],
      "type": "Polygon",
      "coordinates": [
        [-89.33167167,29.96], [-89.25914630,29.96],
        [-89.25914630,30], [-89.33167167,30]
      ]
    }
  ]
}
```

```
    }]
}
```

The preceding code is valid GeoJSON, but it is also a valid Python data structure. You can copy the preceding code sample directly into the Python interpreter as a variable definition and it will evaluate without error, as follows:

```
>>> gc = { "type": "GeometryCollection",
...     "geometries": [
...         { "type": "Point",
...             "coordinates": [
...                 [-89.33,30]]
...         },
...         { "type": "LineString",
...             "coordinates": [
...                 [-89.33,30.3],[-89.36,30.28]],
...         },
...         { "type": "Polygon",
...             "coordinates": [
...                 [-89.33167167,29.96],[-89.25914630,29.96], [
...                     [-89.25914630,30],[-89.33167167,30]],
...             ]
...         }
...     }
>>> gc
{'type': 'GeometryCollection', 'geometries': [{'type': 'Point',
'coordinates': [[-89.33, 30]]}, {'type': 'LineString', 'coordinates':
[[[-89.33, 30.3], [-89.36, 30.28]]]}, {'type': 'Polygon',
'coordinates': [[[[-89.33167167, 29.96], [-89.25914630, 29.96], [-89.25914630, 30], [-89.33167167, 30]]]]}]}
```

Due to its compact size, internet-friendly syntax by virtue of being written in JavaScript, and support from major programming languages, GeoJSON is a key component of leading REST geospatial web APIs, which will be covered later in this chapter. It currently offers the best compromise among the computer resource efficiency of binary formats, the human-readability of text formats, and programmatic utility.

## GeoPackage

We'll briefly mention the GeoPackage format here as it will be covered in *Chapter 3, The Geospatial Technology Landscape*, as well as because it's a type of geodatabase. The GeoPackage format is an OGC open standard on a SQLite file-based database container that is a platform-, vendor-, and software-independent. It's an attempt to get away from all of the issues that are generated from either proprietary data formats or limited data formats.

Next, we'll look at the other major data type, **raster data**.

## Understanding raster data types

Raster data consists of rows and columns of cells or pixels, with each cell representing a single value. The easiest way to think of raster data is as images, which is how they are typically represented by software. However, raster datasets are not necessarily stored as images. They can also be **ASCII text files** or **Binary Large Objects (BLOBs)** in databases.

Another difference between geospatial raster data and regular digital images is their resolution. Digital images express resolution as dots-per-inch if printed in full size. Resolution can also be expressed as the total number of pixels in the image, and are defined as megapixels. However, geospatial raster data uses the ground distance that each cell represents. For example, a raster dataset with a two-foot resolution means that a single cell represents two feet on the ground, which also means that only objects larger than two feet can be identified visually in the dataset.

Raster datasets may contain multiple bands, meaning that different wavelengths of light can be collected at the same time over the same area. Often, this range is from 3-7 bands but it can be several hundred in hyperspectral systems. These bands are viewed individually or swapped in and out as the RGB bands of an image. They can also be recombined into a derivative single-band image using mathematics and then recolored using a set number of classes representing values within the dataset.

Another common application of raster data is in the field of scientific computing, which shares many elements of geospatial remote sensing but adds some interesting twists.

Scientific computing often uses complex raster formats, including **Network Common Data Form (NetCDF)**, **GRIB**, and **HDF5**, which store entire data models. These formats are more like directories in a filesystem and can contain multiple datasets or multiple versions of the same dataset. Oceanography and meteorology are the most common applications of this kind of analysis. An example of a scientific computing dataset is the output of a weather model, where the cells of the raster dataset in different bands may represent different variables' output from the model in a time series.

Like vector data, raster data can come in a variety of formats. The open source raster library known as **Geospatial Data Abstraction Library (GDAL)**, which actually includes the vector OGR library we mentioned earlier, lists over 130 supported raster formats ([http://www.gdal.org/formats\\_list.html](http://www.gdal.org/formats_list.html)). The FME software package supports this many as well. However, just like shapefiles and CAD data, there are a few standout raster formats.

### TIFF files

The **Tagged Image File Format (TIFF)** is the most common geospatial raster format. The TIFF format's flexible tagging system allows it to store any type of data whatsoever in a single file. TIFFs can contain overview images, multiple bands, integer elevation data, basic metadata, internal compression, and a variety of other data that's typically stored in additional supporting files by other formats. Anyone can extend the TIFF format unofficially by adding tagged data to the file structure. This extensibility has benefits and drawbacks. A TIFF file may work fine in one piece of software but fail when it's accessed

in another because the two software packages implement the massive TIFF specification to different degrees. An old joke about TIFFs has a frustrating amount of truth to it: **TIFF** stands for **Thousands of Incompatible File Formats**. The GeoTIFF extension defines how geospatial data is stored. Geospatial rasters stored as TIFF files may have any of the following file extensions: `.tiff`, `.tif`, or `.gtif`.

## JPEG, GIF, BMP, and PNG

JPEG, GIF, BMP, and PNG formats are common image formats in general but can be used for basic geospatial data storage as well. Typically, these formats rely on accompanying the supporting text files for the georeferencing of the information in order to make them compatible with the GIS software, such as WKT, `.prj`, or world files.

The JPEG format is also fairly common for geospatial data. JPEGs have a built-in metadata tagging system, similar to TIFFs, called EXIF. JPEGs are commonly used for geotagged photographs in addition to raster GIS layers. **Bitmap (BMP)** images are used for desktop applications and document graphics. However, JPEG, GIF, and PNG are the formats that are used in web mapping applications, especially for pre-generated server map tiles for quick access via tiled maps.

## Compressed formats

Since geospatial rasters tend to be very large, they are often stored using advanced compression techniques. The latest open standard is the JPEG 2000 format, which is an upgrade of the JPEG format and includes wavelet compression and a few other features, such as georeferencing data. The **Multi-resolution Seamless Image Database (MrSID) ( .sid)** and **Enhanced Compression Wavelet (ECW) ( .ecw)** are two proprietary wavelet compression formats often seen in geospatial contexts.

The TIFF format supports compression, including the **Lempel-Ziv-Welch (LZW)** algorithm. It must be noted that compressed data is suitable as part of a basemap but should not be used for remote sensing processing. Compressed images are designed to look visually correct but often alter the original cell value. Lossless compression algorithms try to avoid degrading the source data, but it's generally considered a bad idea to attempt to perform spectral analysis on data that has been through compression. The JPEG format is designed to be a lossy format that sacrifices data for a smaller file size. It is also commonly encountered, so it is important to remember this fact to avoid invalid results.

## ASCII grids

Another means of storing raster data, often elevation data, is in ASCII Grid files. This file format was created by Esri but has become an unofficial standard supported by most software packages. An ASCII grid is a simple text file containing  $(x, y)$  values as rows and columns. The spatial information for the raster is contained in a simple header. The format of the file is as follows:

```
<NCOLS xxx>
<NROWS xxx>
```

```

<XLLCENTER xxx | XLLCORNER xxx>
<YLLCENTER xxx | YLLCORNER xxx>
<CELLSIZE xxx>
{NODATA_VALUE xxx}
row 1
row 2
.
.
.
row n

```

While not the most efficient way to store data, ASCII Grid files are very popular because they don't require any special data libraries to create or access geospatial raster data. These files are often distributed as .zip files. The header values in the preceding format contain the following information:

- The number of columns
- The number of rows
- The *x* axis cell center coordinate or *x* axis lower-left corner coordinate
- The *y* axis cell center coordinate or *y* axis lower-left corner coordinate
- The cell size in mapping units
- The no-data value (typically, 9,999)

## World files

World files are simple text files that can provide geospatial referencing information to any image externally for file formats that typically have no native support for spatial information, including JPEG, GIF, PNG, and BMP. A world file is recognized by geospatial software due to its naming convention. The most common way to name a world file is by using the raster filename and then altering the extension to remove the middle letter and adding w at the end.

The following table shows some examples of raster images in different formats and the associated world filename based on the convention:

Raster filename	World filename
World.jpg	World.jgw
World.tif	World.tfw
World.bmp	World.bpw
World.png	World.pgw
World.gif	World.gfw

Table 2.2 – Raster world file naming conventions by file format type

The structure of a world file is very simple. It is a six-line text file, as follows:

- **Line 1:** The cell size along the  $x$  axis in ground units
- **Line 2:** The rotation on the  $y$  axis
- **Line 3:** The rotation on the  $x$  axis
- **Line 4:** The cell size along the  $y$  axis in ground units
- **Line 5:** The center  $x$  coordinate of the upper-left cell
- **Line 6:** The center  $y$  coordinate of the upper-left cell

The following is an example of world file values:

```
15.0
0.0
0.0
-15.0
-89,38
45.0
```

The  $(x, y)$  coordinates and the  $(x, y)$  cell size contained in lines 1, 4, 5, and 6 allow you to calculate the coordinate of any cell or the distance across a set of cells. The rotation values are important for geospatial software because remotely sensed images are often rotated due to the data collection platform.

Rotating the images runs the risk of resampling the data and, therefore, data loss, so the rotation values allow the software to account for the distortion. The surrounding pixels outside the image are typically assigned a no-data value and represented as the color black.

The following Landsat 8 satellite image, courtesy of the **U.S. Geological Survey (USGS)** and NASA, from <https://earthexplorer.usgs.gov/>, demonstrates image rotation, where the satellite collection path is oriented from southeast to northeast, but the underlying basemap is north:



Figure 2.5 – A satellite image on a north-up basemap oriented to the satellite orbit path

World files are a great tool when working with raster data in Python. Most geospatial software and data libraries support world files, so they are usually a good choice when it comes to georeferencing.

#### More on world files

You'll find that world files are very useful, but if you use them infrequently, you will forget what the unlabeled contents represent. A quick reference for world files is available at <https://kralidis.ca/gis/worldfile.htm>.

Vector data and raster data are the two most common data types. However, there is another type that is gaining popularity due to the cost of collecting it gradually becoming cheaper. That type is point cloud data, and we'll examine it next.

## What is point cloud data?

Point cloud data is any data collected as the  $(x, y, z)$  location of a surface point based on some sort of focused energy return. This can be created using lasers, radar waves, acoustic sounding, or other waveform generation devices. The spacing between points is arbitrary and dependent on the type and position of the sensor collecting the data.

In this book, we will primarily be concerned with LIDAR data and radar data. Radar point cloud data is typically collected on space missions, while LIDAR is typically collected by terrestrial or airborne vehicles. Conceptually, these types of data are similar.

### LIDAR

LIDAR uses powerful laser range-finding systems to model the world with very high precision. The term LIDAR, or lidar, is a combination of the words light and radar. Some people claim it also stands for **Light Detection and Ranging**. LIDAR sensors can be mounted on aerial platforms, including satellites, airplanes, or helicopters. They can also be mounted on vehicles for ground-based collection.

Due to the high-speed, continuous data collection provided by LIDAR, and a wide field of view – often 360-degree sensors – LIDAR data doesn't typically have a rectangular footprint the way other forms of raster data do. LIDAR datasets are usually called point clouds because the data is a stream of  $(x,y,z)$  locations, where  $z$  is the distance from the laser to a detected object and the  $(x,y)$  values are the projected location of the object calculated from the location of the sensor.

The following image, courtesy of USGS, shows a point cloud LIDAR dataset in an urban area using a terrestrial sensor, as opposed to an aerial one. The colors are based on the strength of the laser's energy return, with red areas being closer to the LIDAR sensor and green areas farther away, which can give a precise height to within a few centimeters.

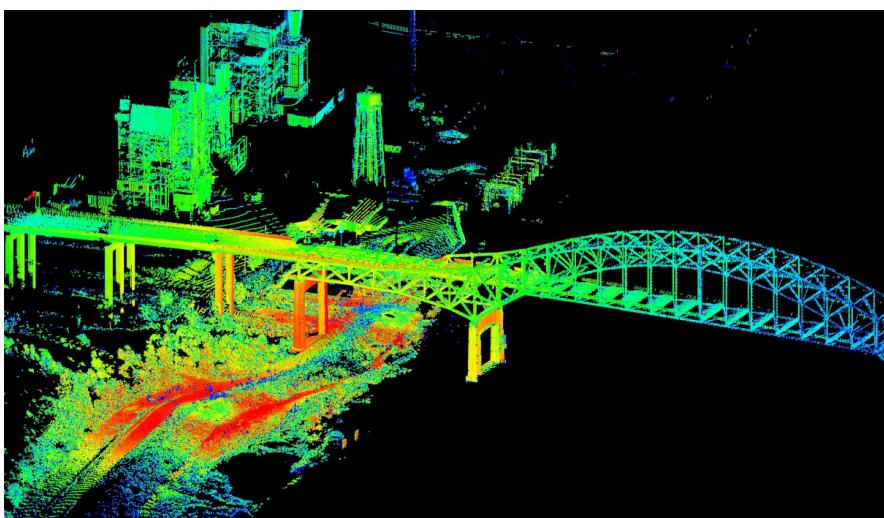


Figure 2.6 – A LIDAR point cloud color-coded by the intensity of the return

The most common data format for LIDAR data is the **LIDAR Exchange Format (LAS)**, which is a community standard. LIDAR data can be represented in many ways, including a simple text file with one  $(x, y, z)$  tuple per line. Sometimes, LIDAR data can be colorized using image pixel colors that have been collected at the same time. LIDAR data can also be used to create 2D elevation rasters.

This technique is the most common use for LIDAR in geospatial analysis. Any other use requires specialized software that allows the user to work in 3D. In that case, other geospatial data cannot be combined with the point cloud data.

## More realistic geospatial models with 3D data

**Three-dimensional (3D)** geospatial data is a type of data that represents geographic information in three dimensions, meaning it includes information about height, width, and depth. The purpose of 3D geospatial data is to provide a more complete and accurate representation of the real world by adding a third dimension to traditional two-dimensional maps. Until recently, 3D data wasn't used often because it took many times more computer processing power and memory to analyze and render graphically than 2D geospatial data. But as computing power has significantly increased and cloud processing has become more common, so has the use of 3D data, which is better supported by modern technology.

One of the most popular tools for viewing and interacting with 3D data is Google Earth, which we'll see in *Chapter 3, The Geospatial Technology Landscape*. We examined one of the most common types of 3D data, LIDAR data, earlier in this chapter.

Common uses of 3D geospatial data include the following:

- **Urban planning and design:** 3D geospatial data can be used to create digital models of cities and towns, allowing urban planners and architects to visualize and plan new buildings, roads, and other infrastructure
- **Disaster management:** 3D geospatial data can help emergency responders and relief workers better understand the terrain and infrastructure of an affected area, enabling them to plan and execute disaster response efforts more effectively
- **Environmental management:** 3D geospatial data can be used to monitor changes in the natural environment, such as deforestation, erosion, and land use, helping conservationists and environmental managers make informed decisions about resource management

Three-dimensional data or 3D data in its simplest form is any data format with an  $(x,y,z)$  location including LIDAR data and shapefiles with a  $z$  attribute. But it can also include a variety of 3D modeling formats that programs including Google Earth and ArcGIS can display. Point cloud data can also be converted to surfaces such as **Triangular Irregular Networks (TINs)**. 3D data is used mainly for advanced visualizations and is not part of everyday geospatial analysis, but it's worth mentioning as 3D world models are becoming increasingly more common.

## What are web services?

Geospatial web services allow users to perform data discovery, data visualization, and data access across the web. Web services are usually accessed by applications based on user input, such as zooming in to an online map or searching a data catalog. The most common protocols are **Web Map Service (WMS)**, which returns a rendered map image, and **Web Feature Service (WFS)**, which typically returns GML, which was mentioned in this chapter's introduction.

Many WFS services can also return KML, JSON, zipped shapefiles, and other formats. These services are called through HTTP GET requests. The following URL is an example of a WMS GET request, which returns a map image of an island in the Gulf of Mexico that is 1,125 pixels wide by 889 pixels tall and has an EPSG code of 4326 (WGS 84 Geographic):

```
https://basemap.nationalmap.gov:443/arcgis/services/USGSImagery  
Only/MapServer/WmsServer?SERVICE=WMS&VERSION=1.3.0&REQUEST=Get  
Map&BBOX=30.1870873913039155,-89.15109993139789424,30.25736630  
077669176,-89.062164303549892&CRS=EPSG:4326&WIDTH=1125&HEIGHT  
=889&LAYERS=0&STYLES=&FORMAT=image/jpeg&DPI=96&MAP_RESOLUTION=  
96&FORMAT_OPTIONS=dpi:96
```

Web services are rapidly evolving. The Open GIS Consortium is adding new standards for sensor networks and other geospatial contexts. **REpresentational State Transfer (REST)** services are also commonly used. REST services use simple URLs to make requesting data very easy to implement in nearly any programming language by tailoring URL parameters and their values accordingly. It enables communication between clients (web applications) and servers (GIS servers). And by either using URL parameters or sending data via the request body, we can specify the parameters to get qualified results. Almost every programming language has robust HTTP client libraries that are capable of using REST services.

These REST services can return many types of data, including images, XML, or JSON. There is no overarching geospatial REST standard yet, but the OGC has been working on one for quite some time as part of their OpenAPI initiative. Esri has created a working implementation that is currently widely used.

### More on web services

The following URL is an example of an Esri geospatial REST service that would return KML based on a weather radar image layer. You can add this URL to Google Earth as a network link, or you can download it as compressed KML (KMZ) in a browser to import it into another program:

```
https://idpgis.ncep.noaa.gov/arcgis/rest/services/NWS_  
Observations/radar_base_reflectivity/MapServer/generateKml?  
docName=NWSRadar&layers=0&layerOptions=separateImage
```

You can find tutorials on the myriad of OGC services at <http://opengeospatial.github.io/e-learning/fundamental-concepts/text/index.html>.

You can find more about the OGC OpenAPI initiative at <https://ogcapi.ogc.org/>.

Now, we'll move from individual file formats to powerful geodatabases that can consolidate data through a single API.

## Understanding geospatial databases

A geospatial database, or geodatabase, refers to an entire category of file formats, data schemas, and even software. In *Chapter 3, The Geospatial Technology Landscape*, we'll cover geodatabases as software packages, formally known as database management systems. But in this section, we'll describe their attributes as file formats. Geodatabases historically stored only vector data, though modern geodatabases are well-suited to raster data management as well.

Geodatabases can exhibit all of the common traits we noted previously. This information is stored in the database, in what we call the database model. A very popular model is the traditional relational model, which uses tables of rows and columns. Each row and column combination is called a cell. Rows can be related to another table to link information using a designated column where each cell becomes a key referencing a cell in another table that then links the rows together.

The actual names of the columns and the relationships among data constitute the data definition. At a minimum, geodatabases associate a geometry description with attributes about the object the geometry represents. Single points are often represented by *x* and *y* columns. However, polygons and polylines have arbitrary numbers of points. This means that geodatabases often store geometry information as a BLOB using a format standard known as **Well-Known Binary**, or **WKB**.

The attribute information is usually defined as data types such as integers, floating-point decimal numbers, strings, or dates. The table may also include projection information for map display, as well as a spatial indexing column to speed up searching and geospatial comparisons. A geodatabase may also have another related table in order to link detailed metadata about the geospatial data.

Large geospatial raster datasets are rarely stored directly in the database. Typically, the raster data is stored on disk with a name, and a filesystem reference is stored in the database that points to the raster data. A geodatabase may also store a geometry column representing the ground footprint of the raster data, which can then be used as a proxy for geospatial operations.

## Sharing data with interchange formats

In the early days of geospatial analysis, different software packages had their own proprietary file formats. When analysts needed to share data with people using a different system, they would have to convert it to an interchange format that had an open specification that different software packages could convert into their own proprietary format. Early examples of interchange formats included Esri Shapefiles and the ArcInfo Interchange format, which had the extension E00. The limited number of formats to exchange data and the limitations of those formats, such as limits on file size, made sharing inefficient and inconvenient. The Open GIS Consortium and the open source software community in general worked to create new standards to increase the number of interchange formats using open

specifications. The tag-based XML formats described earlier in this chapter, GeoJSON, and the file-based geodatabases including GeoPackage and Esri's File Geodatabase format have revolutionized data sharing and availability because there are so many options available. Text-based formats such as GeoJSON aren't the most efficient way to store data, but their relative simplicity and the speed of modern computers have made them ubiquitous.

The following figure shows clustered points of earthquake locations stored in the GeoJSON interchange format rendered by the GitHub code repository:

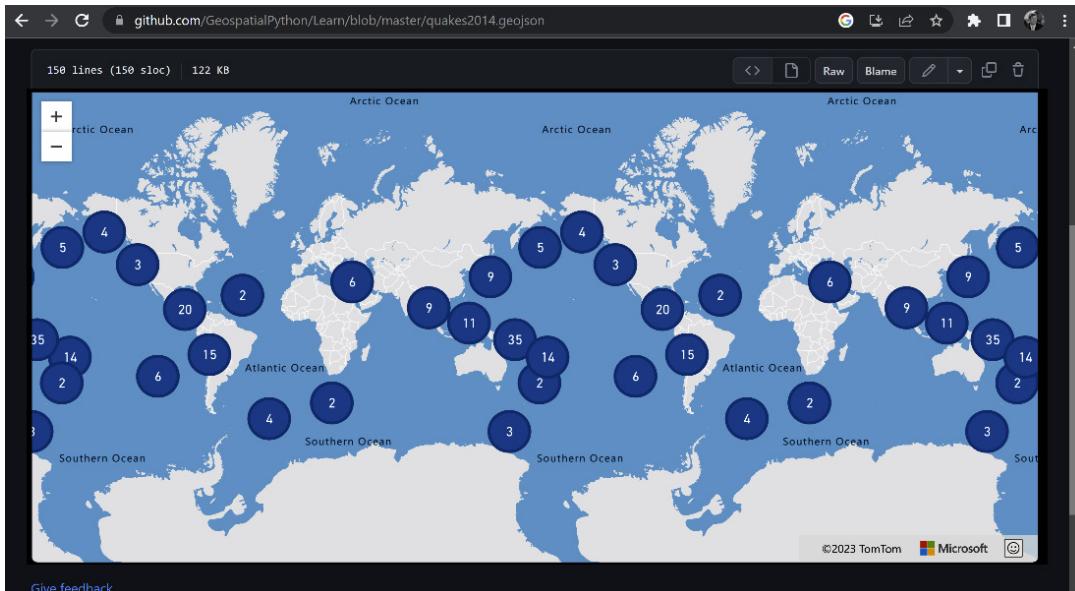


Figure 2.7 – Clustered points on earthquake locations from GitHub

Interchange formats like this GeoJSON sample make it easier for other software, such as GitHub.com in this example, to easily work with geospatial data as opposed to more complicated legacy formats.

## Introducing spatiotemporal data

Spatiotemporal data is geospatial data that contains a time series element. Time series analysis shows how features evolve over time within a space. It can be used in a variety of contexts but is most frequently used in sociological, demographic, environmental, and meteorology/climate studies. For example, you might analyze rainfall data in a city for a period of time. It can also be used to visualize population shifts in a country. Time series data allows for trend analysis or giving data a broader context.

The following figure shows a jogger's workout visualized from Garmin GPX data with each point on the line representing the runner at a different location along the route over time. The speed at each given time is also attached to each point.

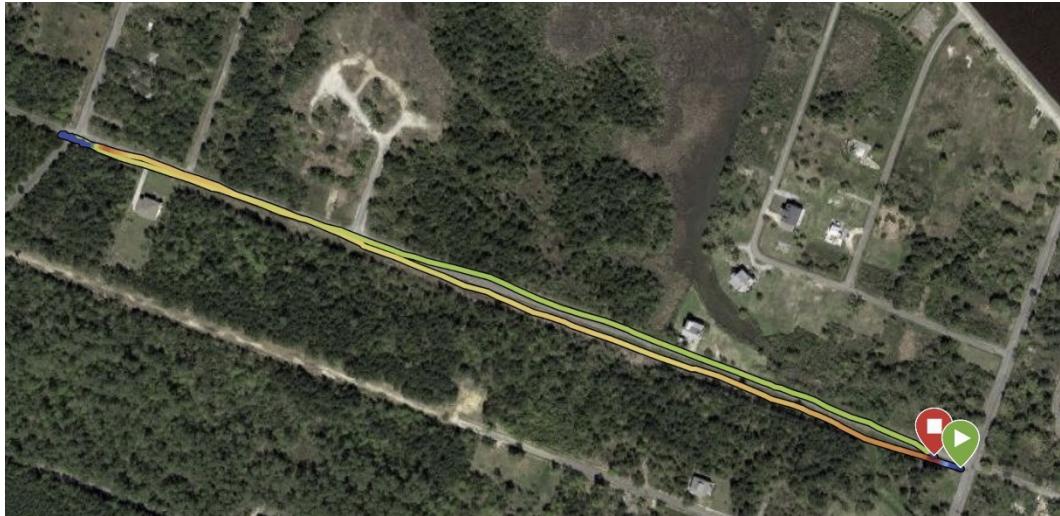


Figure 2.8 – A spatiotemporal data example showing speed over time on a road

The following graph shows the speed values from this dataset over time:

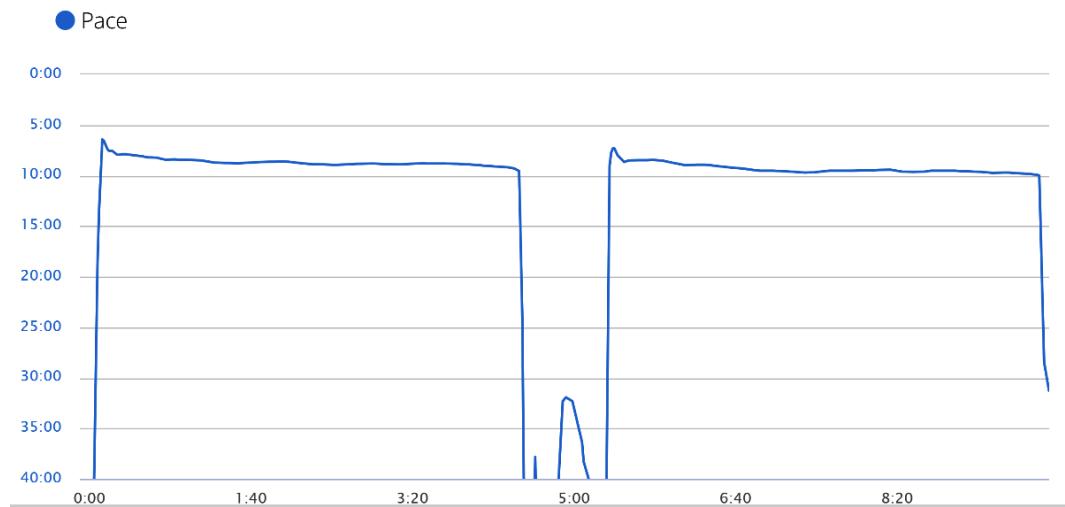


Figure 2.9 – A graph of speed versus time without the spatial component

Spatiotemporal data isn't a data format but rather a type of data. Any data format that can represent a location and store a time value can store spatiotemporal data. Nearly all formats described in this chapter can represent temporal data, with the exception of simple image formats such as JPEG, GIF, BMP, and PNG.

## Summary

In this chapter, we discussed the different types of geospatial data you will encounter, including vector, raster, and temporal data types. We learned about the supporting formats that usually accompany geospatial datasets to make them easier to work with, including spatial indexes and metadata. We examined the storage convenience and processing power of geodatabases as well as the delivery efficiency of geospatial web services. We also discussed the immersive visualizations and in-depth modeling possible with 3D geospatial data.

You now have the background needed to work with common types of geospatial data. You also know about the common traits of geospatial datasets that will allow you to evaluate unfamiliar types of data and identify key elements that will drive you toward which tools to use when interacting with this data.

In the next chapter, we'll examine the modules and libraries that you can use to work with geospatial datasets. We will learn about the geospatial technology ecosystem, which consists of thousands of software libraries and packages. We will also understand the hierarchy of geospatial software and how it allows you to quickly comprehend and evaluate any geospatial tool.

## Questions

1. What are the two major types of geospatial data?
2. What are the algorithms called that organize geospatial data for more efficient search and display?
3. What are the three primary attributes that make up location data in a point cloud?

## Further reading

- <https://www.safe.com/what-is/geospatial-data/>
- <https://guides.library.brandeis.edu/c.php?g=990410&p=7164688>
- <https://www.oracle.com/autonomous-database/what-is-geospatial-database/>
- <https://1spatial.com/news-events/2022/what-is-geospatial-data-and-how-is-it-used/>



# 3

## The Geospatial Technology Landscape

The geospatial technology ecosystem consists of hundreds of software libraries and packages. This vast array of choices can be overwhelming for newcomers to geospatial analysis. The secret to learning geospatial analysis quickly is understanding the handful of libraries and packages that really matter. Most software, both commercial and open source, is derived from these critical packages. Understanding the ecosystem of geospatial software and how it's used allows you to quickly comprehend and evaluate any geospatial tool.

Geospatial libraries can be assigned to one or more of the following high-level core capabilities, which they implement to some degree. We will be learning about the following capabilities in this chapter:

- Understanding data access
- Understanding computational geometry
- Image processing (we'll focus on the common data access libraries)
- Understanding desktop tools (including visualization)
- Understanding metadata management

In this chapter, we'll examine the packages that have had the largest impact on geospatial analysis, and also those that you are likely to frequently encounter. However, as with any filtering of information, you are encouraged to do your own research and draw your own conclusions.

The following websites offer more information on software that is not included in this chapter:

- Wikipedia list of GIS software: [https://en.wikipedia.org/wiki/List\\_of\\_geographic\\_information\\_systems\\_software](https://en.wikipedia.org/wiki/List_of_geographic_information_systems_software)
- OSGeo project list and incubator projects: <http://www.osgeo.org>

The image processing software capability is for remote sensing. However, this category of software is very fragmented, containing dozens of software packages that are rarely integrated into derivative software. Most image processing software for remote sensing is based on the same data access libraries, with custom image processing algorithms implemented on top of them.

Take a look at the following examples of these types of software, which include both open source and commercial packages:

- **Open Source Software Image Map (OSSIM)**
- **Geographic Resources Analysis Support System (GRASS)**
- **Orfeo ToolBox (OTB)**
- **ERDAS IMAGINE**
- **ENVI**

By the end of this chapter, you'll understand the key pieces of both the proprietary and open source software in the geospatial industry used to view and analyze the data formats we learned about in *Chapter 2, Learning about Geospatial Data*.

## Technical requirements

The following is a list of technical requirements for this chapter:

- Python 3.6 or higher
- RAM: Minimum 6 GB (Windows), 8 GB (macOS); recommended 8 GB
- Storage: Minimum 7200 RPM SATA with 20 GB of available space; recommended SSD with 40 GB of available space
- Processor: Minimum Intel Core i3 2.5 GHz; recommended Intel Core i5 common traits

## Understanding data access

As described in *Chapter 2, Learning about Geospatial Data*, geospatial datasets are typically large, complex, and varied. This challenge makes libraries that efficiently read, and in some cases write, this data essential to geospatial analysis. Without access to data, geospatial analysis cannot begin.

Furthermore, accuracy and precision are key factors in geospatial analysis. An image library that resamples data without permission, or a computational geometry library that rounds a coordinate by even a couple of decimal places, can adversely affect the quality of the analysis. Also, these libraries must manage memory efficiently. A complex geospatial process can last for hours, or even days.

If a data access library has a memory fault, it can delay an entire project or even an entire workflow, involving dozens of people who are dependent on the output of that analysis.

Data access libraries such as **Geospatial Data Abstraction Library (GDAL)** are mostly written in either C or C++ for speed and cross-platform compatibility. Speed is important due to the typically large size of geospatial datasets. However, you will also see many packages written in Java. When it's well written, pure Java can approach speeds that are acceptable for processing large vector or raster datasets, and that are usually acceptable for most applications.

The following concept map shows the major geospatial software libraries and packages and how they are related. The libraries in bold represent root libraries that are actively maintained and are not significantly derived from any other libraries. These root libraries represent geospatial operations, which are rather difficult to implement, and the vast majority of people choose to use one of these libraries, rather than create a competing one. As you can see, a handful of libraries make up a disproportionate amount of geospatial analysis software. The following diagram is by no means exhaustive. In this book, we'll discuss only the most commonly used packages.

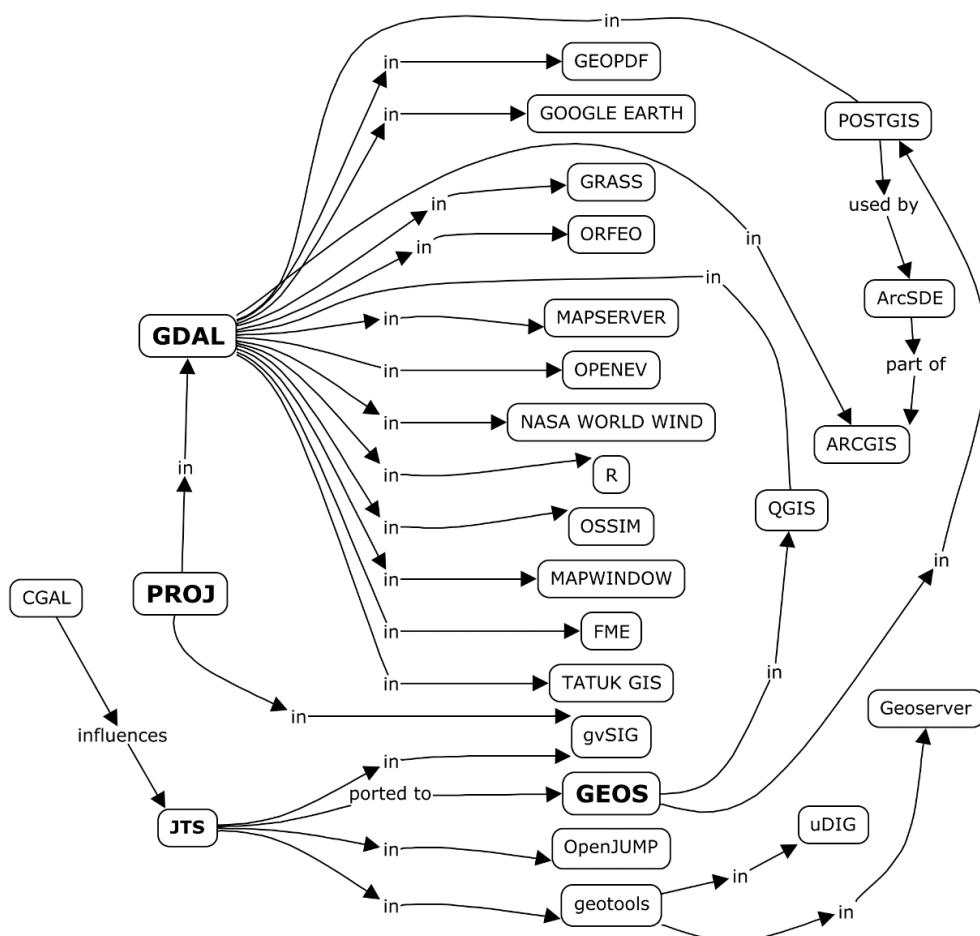


Figure 3.1 – Concept map illustrating the connections between key pieces of geospatial software, most of which revolve around GDAL, PROJ, GEOS, and JTS

Note that Python is not mentioned in this diagram. That's because Python can, in one way or another, interact with all of these libraries and software.

The **GDAL**, **GEOS** (short for **Geometry Engine – Open Source**), and **PROJ** libraries are the heart and soul of the geospatial analysis community on both the commercial and open source sides. It is important to note that these libraries are all written in C or C++. There is also significant work done in Java in the form of the **GeoTools** and **Java Topology Suite (JTS)** core libraries, which are used across a range of desktops, servers, and mobile software. Given that there are hundreds of geospatial packages available, with nearly all of them relying on these libraries to do anything meaningful, you'll begin to get an idea of the complexity of geospatial data access and computational geometry. Compare this software domain to that of a simpler category of software such as text editors, which return over 35,459 options when searched on the open source project site GitHub (<https://github.com/>).

Geospatial analysis is a truly worldwide community, with significant contributions to the field coming from every corner of the globe. But as you learn more about the heavy-hitting packages at the center of the software landscape, you'll see that these programs tend to come from Canada or are contributed to heavily by Canadian developers.

Credited as the birthplace of modern GIS, geospatial analysis is a matter of national pride. Also, the Canadian government and the public–private GeoConnections program have invested heavily in research and companies, both to fuel the industry for economic reasons, and out of necessity – to better manage the country's vast natural resources and the needs of its population.

## **GDAL**

GDAL does the most heavy-lifting tasks in the geospatial industry. The GDAL website lists over 80 pieces of software using the library, and this list is by no means complete. Many of these packages are industry-leading, open source, and commercial tools. This list doesn't include the hundreds of smaller projects and individual analysts who use the library for geospatial analysis. GDAL also includes a set of command-line tools that can do a variety of operations without any programming.

### **More about GDAL**

A list of projects using GDAL can be found at the following URL: [https://gdal.org/software\\_using\\_gdal.html](https://gdal.org/software_using_gdal.html).

### ***GDAL and raster data***

GDAL provides a single, abstract data model for the vast array of raster data types that are found in the geospatial industry. It consolidates unique data access libraries for different formats and provides a common API for reading and writing data. Before the developer Frank Warmerdam created GDAL in the late 1990s, each data format required a separate data access library with a different API in order to read data, or in the worst-case scenario, developers often wrote custom data access routines.

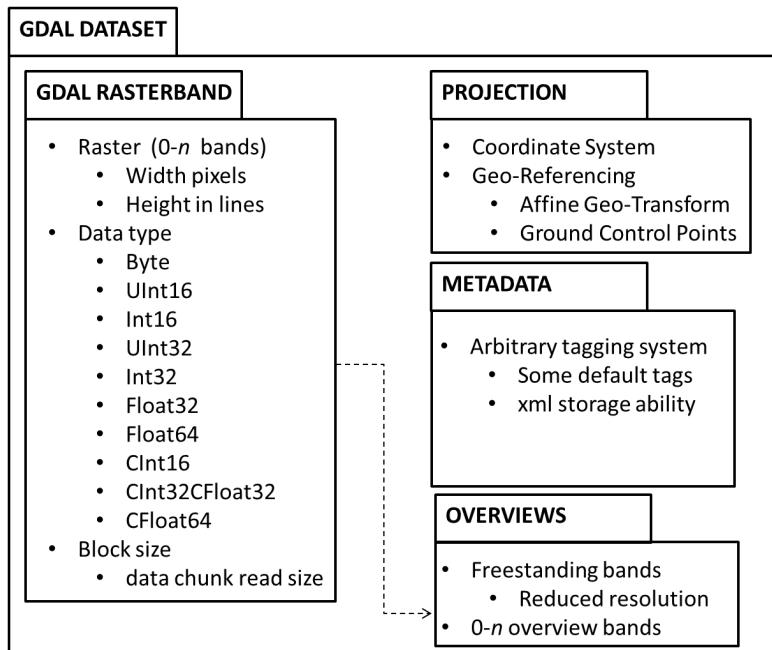


Figure 3.2 – Chart describing the GDAL data model used to access dozens of different raster data formats

In the preceding software concept map, you can see that GDAL has had the greatest impact of any single piece of geospatial software. Combine GDAL with its sister library, OGR, for vector data, and the impact almost doubles. The PROJ library has also had a tremendous impact, but it is usually accessed via OGR or GDAL.

### ***GDAL and vector data***

In addition to raster data, GDAL lists at least partial support for over 70 vector data formats. Part of the success of the GDAL package is the X11/MIT open source license. This license is both commercial and open-source-friendly. The GDAL library can be included in proprietary software without revealing the proprietary source code to users.

GDAL has the following vector capabilities:

- Uniform vector data and modeling abstraction
- Vector data reprojection
- Vector data format conversion
- Attribute data filtering
- Basic geometry filtering including clipping and point-in-polygon testing

GDAL has several command-line utility programs, which demonstrate its capability for vector data. This capability can also be accessed through its programming API. The following diagram outlines the GDAL vector architecture:

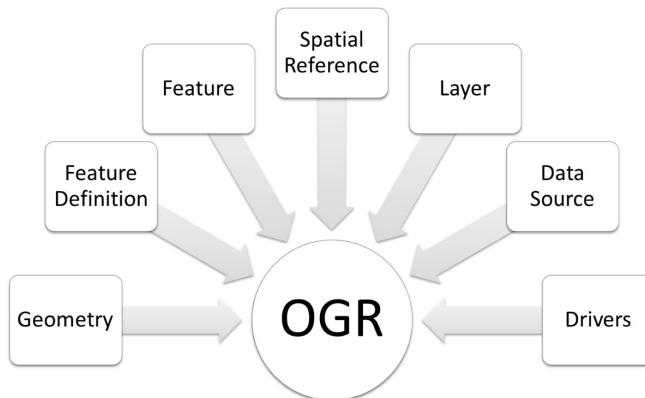


Figure 3.3 – Diagram illustrating the key elements that make up the OGR package

The GDAL vector architecture is fairly concise, considering this model is able to represent over 70 different data formats. It's elegantly broken down into some key software abstractions that make up the architecture:

- **Geometry:** This object represents the **Open Geospatial Consortium (OGC) Simple Features Specification** data model for points, linestrings, polygons, geometry collections, multipolygons, multipoints, and multilinestrings
- **Feature Definition:** This object contains the attribute definitions of a group of related features
- **Feature:** This object ties the **Geometry** and **Feature Definition** information together
- **Spatial Reference:** This object contains an OGC spatial reference definition
- **Layer:** This object represents features that are grouped as layers within a data source
- **Data Source:** This object is the file or database object that is accessed by GDAL
- **Drivers:** This object contains the translators for the 70-plus data formats that are available to GDAL

This architecture works smoothly, with one minor quirk – the layer concept is used even for data formats that only contain a single layer. For example, shapefiles can only represent a single layer. But, when you access a shapefile using GDAL, you must still invoke a new layer object using the base name of the shapefile without a file extension. This design feature is only a minor inconvenience, heavily outweighed by the power that GDAL provides.

## PDAL

PDAL stands for **Point Data Abstraction Library** and is modeled after some of the core concepts of GDAL but specializes in point cloud data in a way that is unique. PDAL is open source software that helps to efficiently manage very large point cloud datasets. PDAL allows users to read, write, and manipulate various types of point cloud data, such as LiDAR or photogrammetry data, and transform it into different formats or projections. It also provides a wide range of processing and filtering options, such as noise removal or ground extraction, which can be configured to suit specific needs. With its modular architecture, PDAL can be easily integrated into different workflows and can be used both from the command line and through various programming languages. It has a pipeline framework that is based on `geojson`, allowing users to create complex workflows without any programming. Like GDAL, PDAL could be categorized in this chapter as both a data access library as well as a data processing library.

Now, let's go beyond accessing the data, to using it for analysis.

## Understanding computational geometry

Computational geometry encompasses the algorithms that are needed to perform operations on vector data. The field is very old in computer science; however, most of the libraries used for geospatial operations are separate from computer graphics libraries because of geospatial coordinate systems. As described toward the end of *Chapter 1, Learning about Geospatial Analysis with Python*, computer screen coordinates are almost always expressed in positive numbers, while geospatial coordinate systems often use negative numbers when they're moving west and south.

Several different geospatial libraries fit into this category, but they also serve a wide range of uses, from spatial selection to rendering. It should be noted that some features of GDAL that were described previously move it beyond the category of data access, and into the realm of computational geometry. But it was included in the former category because that is its primary purpose.

Computational geometry is a fascinating subject. When writing a simple script to automate a geospatial operation, you inevitably need a spatial algorithm. The question then arises, “*Do you try to implement this algorithm yourself, or do you go through the overhead of using a third-party library?*”. The choice is always deceptive because some tasks are visually easy to understand and implement, some look complex but turn out to be easy, and some are trivial to comprehend but extraordinarily difficult to implement. One such example is a geospatial buffer operation.

The concept is easy enough, but the algorithm turns out to be quite difficult. The following libraries in this section are the major packages that are used for computational geometry algorithms.

## The PROJ projection library

**U.S. Geological Survey (USGS)** analyst Gerald Evenden created what is now known as the PROJ projection library in the mid-1990s while working at the USGS. Since then, it has become a project of the **Open Source Geospatial Foundation (OSGeo)**, with contributions from many other developers. PROJ accomplishes the Herculean task of transforming data among thousands of coordinate systems. The math that is needed to convert points among so many coordinate systems is extremely complex. No other library comes close to the capability of PROJ. That fact and the routine that is needed by applications to convert datasets from different sources to a common projection make PROJ the undisputed leader in this area.

The following plot is an example of how specific the projections that are supported by PROJ can be. This plot from <https://calcofi.org> represents the line/station coordinate system of the **California Cooperative Oceanic Fisheries Investigations (CalCOFI)** program pseudo-projection, which is used only by NOAA (short for **National Oceanic and Atmospheric Administration**), the University of California Scripps Institution of Oceanography, and the California Department of Fish and Wildlife to collect oceanographic and fisheries data over the last 60 years along the California coastline.

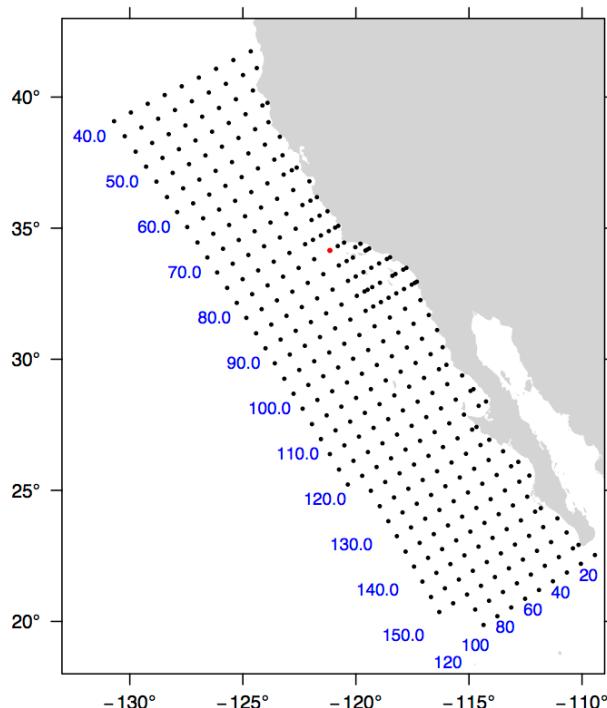


Figure 3.4 – Map in a projection used by a single agency along a specific area of the coast of California, USA, demonstrating just how specific and localized projections can be

PROJ uses a simple syntax that is capable of describing any projection, including custom, localized ones, as shown in the previous plot. PROJ can be found in virtually every major GIS package, providing reprojection support, and it also has its own command-line tools.

It is available through GDAL for vector and raster data. However, it is often useful to access the library directly, because it gives you the ability to reproject individual points. Most of the libraries that incorporate PROJ only let you reproject entire datasets.

### More on PROJ

For more information on PROJ, visit <https://proj.org>.

## CGAL

The **Computational Geometry Algorithms Library (CGAL)**, originally released in the late 1990s, is a robust and well-established open source computational geometry library. It was not specifically designed for geospatial analysis but is commonly used in the field.

CGAL is often referenced as a source for reliable geometry processing algorithms. The following diagram from the *CGAL User and Reference Manual* provides a visualization of one of the often-referenced algorithms from CGAL, called a polygon straight skeleton, which is needed to accurately grow or shrink a polygon:

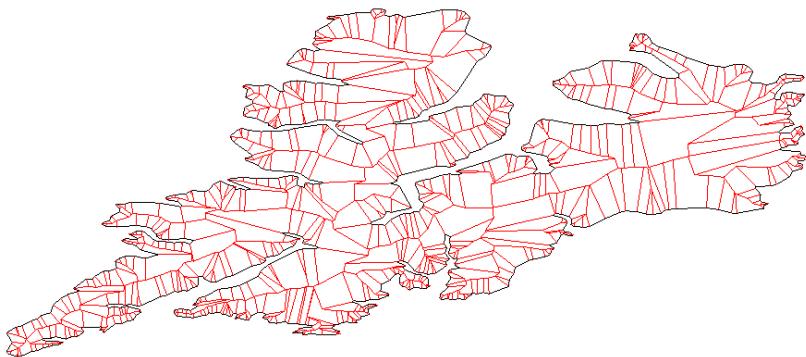


Figure 3.5 – A “straight skeleton” algorithm output of a polygon shows the line structures a computer will use to shrink or grow a polygon accurately

The straight skeleton algorithm is complex and important because shrinking or growing a polygon isn't just a matter of making it bigger or smaller. The polygon actually changes shape. As a polygon shrinks, non-adjacent edges collide and eliminate connecting edges. As a polygon grows, adjacent edges separate, and new edges are formed to connect them. This process is key to the buffering of geospatial polygons. The following diagram, also from the *CGAL User and Reference Manual*, shows this effect using insets on the preceding polygon:

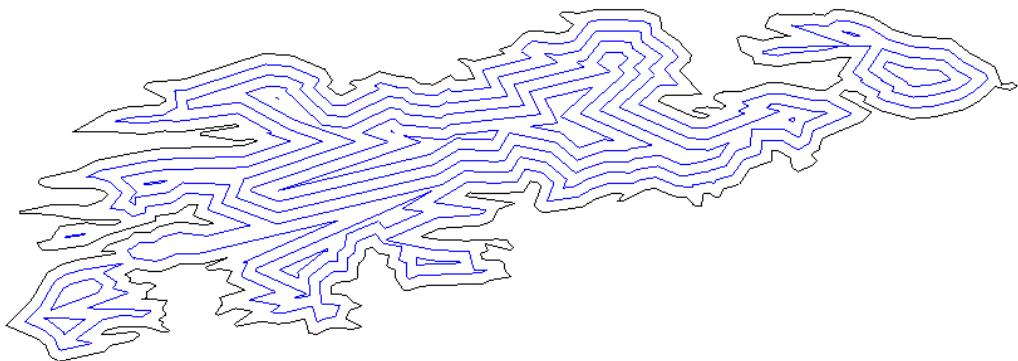


Figure 3.6 – Polygon insets showing that shrinking and growing a polygon actually requires changing its shape in a way that is still recognizable as the size changes

#### More about CGAL

More information on CGAL can be found online at <http://www.cgal.org/>.

## JTS

JTS is a geospatial computational geometry library that is written in 100% pure Java. JTS separates itself from other computational geometry libraries by implementing the OGC Simple Features specification for SQL. Interestingly, other developers have ported JTS to other languages, including C++, Microsoft .NET, and even JavaScript.

JTS includes a fantastic test program called the JTS TestBuilder, which provides a GUI to test functions without setting up an entire program. One of the most frustrating aspects of geospatial analysis concerns bizarre geometry shapes that break algorithms that work most of the time. Another common issue is unexpected results due to tiny errors in data such as polygons that intersect themselves in very small areas that are not easily visible. The JTS TestBuilder lets you interactively test JTS algorithms to verify data, or just to visually understand a process, as shown here:

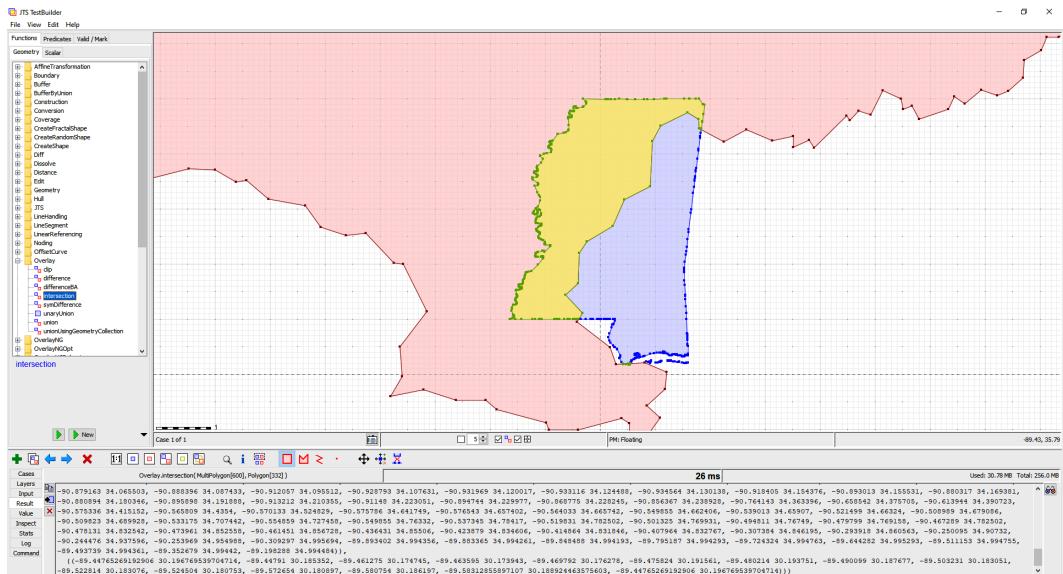


Figure 3.7 – The JTS GUI test program showing an intersection between two polygons

This tool is handy even if you are using one of the several ports of JTS to another language. Even though Vivid Solutions, the creator of JTS, stopped releasing new versions in December 2006, the development was taken over by the Eclipse Foundation and it is actively maintained.

### More about JTS

The JTS home page is available at <https://locationtech.github.io/jts>.

## GEOS

GEOS is the C++ port of the JTS library that was explained previously. It is mentioned here because this port has had a much larger impact on geospatial analysis than the original JTS. The C++ version can be compiled on many platforms, as it avoids any platform-specific dependencies. Another factor responsible for the popularity of GEOS is that a fair amount of infrastructure exists to create automated or semi-automated bindings to various scripting languages, including Python. Another factor is that the majority of geospatial analysis software is written in C or C++. The most common use of GEOS is through other APIs that include GEOS as a library.

GEOS provides the following capabilities:

OGC Simple Features	Within	Distance	Simplify	OGC Well-Known Text (WKT) and Well-Known Binary (WKB) input/output
Geospatial predicate functions	Contains	Intersection	Polygon assembly	C and C++ API
Intersects	Overlaps	Symmetric difference	Polygon validation	Thread safety
Touches	Equals	Convex hull	Area	
Disjoint	Covers	Envelope	Length	
Crosses	Union	Buffer	Spatial indexing	

Table 3.1 – GEOS spatial algorithms

GEOS can be compiled with GDAL to use all of its capabilities.

#### More about GEOS

More information on GEOS can be found online at <https://libgeos.org/>.

## PostGIS

As far as open source geospatial databases go, PostGIS is the most commonly used spatial database. PostGIS is essentially a module on top of the well-known PostgreSQL relational database. Much of the power of PostGIS comes from the previously mentioned GEOS library. Like JTS, it also implements the OGC Simple Features specification for SQL. This combination of computational geometry ability in a geospatial context puts PostGIS in a category of its own.

PostGIS allows you to execute both attribute and spatial queries against a dataset. Recall the point from *Chapter 2, Learning about Geospatial Data*, that a typical spatial dataset comprises multiple data types including geometry, attributes (one or more columns of data in a row), and in most cases, indexing data. In PostGIS, you can query attribute data as you would any database table using SQL.

This capability is not surprising, as attribute data is stored in a traditional database structure. However, you can also query geometry using SQL syntax. Spatial operations are available through SQL functions, which you include as part of your queries. The following sample PostGIS SQL statement creates a 14.5 km buffer around the state of Florida:

```
SELECT ST_Buffer(the_geom, 14500) FROM usa_states
WHERE state = 'Florida'
```

The `FROM` clause designates the `usa_states` layer as the location of the query. We filter that layer by isolating Florida in the `WHERE` clause. `Florida` is a value in the `state` column of the `usa_states` layer. The `SELECT` clause performs the actual spatial selection on the geometry of Florida that is normally contained in the `the_geom` column using the PostGIS `ST_Buffer()` function. The `the_geom` column is the geometry column for the PostGIS layer in this instance. The `ST` abbreviation in the function name stands for spatial type. The `ST_Buffer()` function accepts a column containing spatial geometries and a distance in the map units of the underlying layer.

The map units in the `usa_states` layer are expressed in meters, so 14.5 km would be 14,500 meters in the preceding example. Recall the point from *Chapter 1, Learning about Geospatial Analysis with Python*, that buffers like this query are used for proximity analysis. It just so happens that the Florida state water boundary expands 9 nautical miles, or approximately 14.5 km into the Gulf of Mexico from the state's western and northwestern coastlines.

The following map shows the official Florida state water boundary as a dotted line, which is labeled on the map:

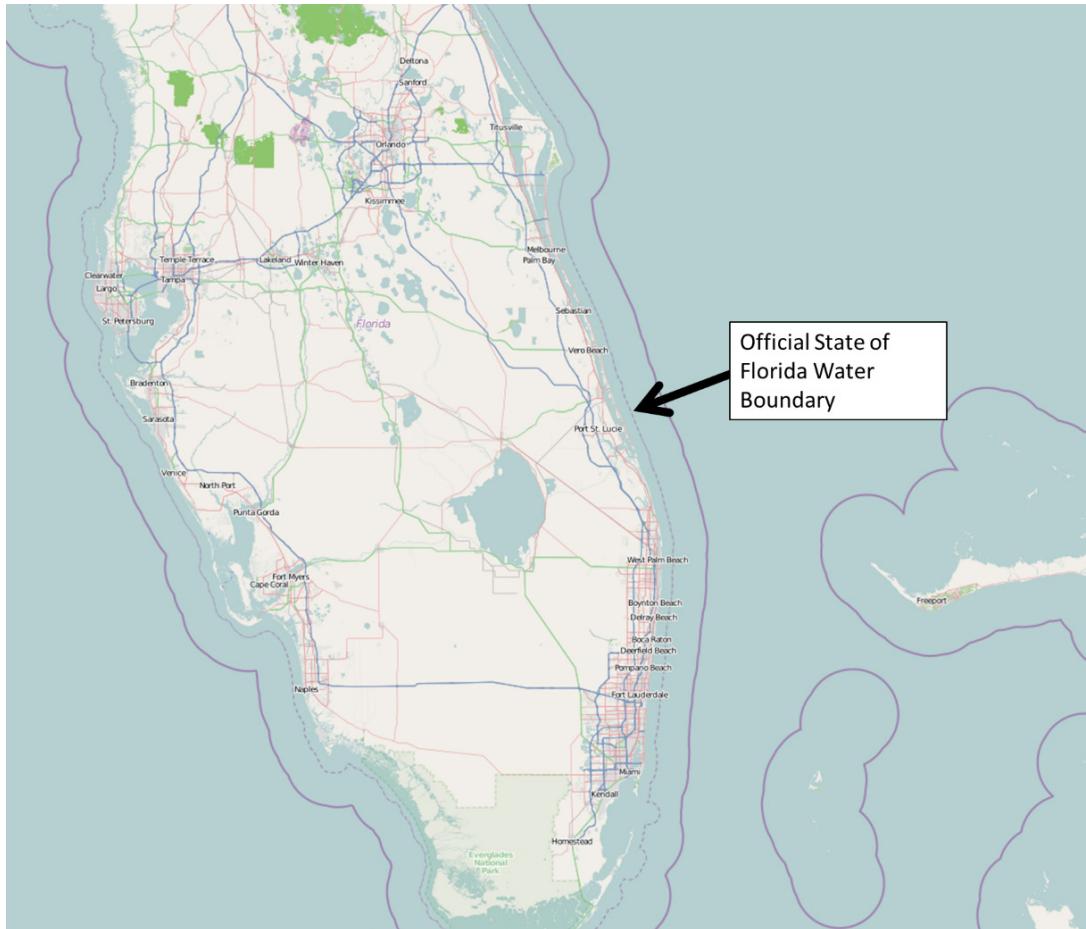


Figure 3.8 – Map showing the state water boundary of the state of Florida, US, which we'll use to perform a buffer operation in PostGIS

After applying the 9-nautical-mile buffer, you can see on the following map that the result, highlighted in orange, is quite close to the official legal boundary, which is based on detailed ground surveys:

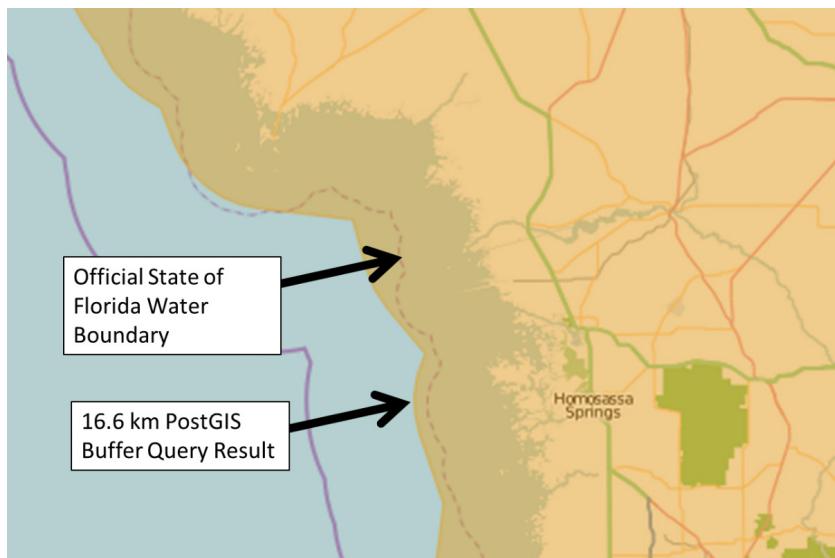


Figure 3.9 – The shaded polygon is the result of a PostGIS spatial buffer query, which tries to approximate the state water boundary

Currently, PostGIS maintains the following feature set:

- Geospatial geometry types, including points, linestrings, polygons, multipoints, multilinestrings, multipolygons, and geometry collections, which can store different types of geometries, including other collections of spatial functions for testing geometric relationships (for example, point-in-polygon or unions)
- Spatial functions for deriving new geometries (for example, buffers, intersections, and reprojection)
- Spatial measurements including perimeter, length, and area
- Spatial indexing using the R-tree algorithm
- A basic geospatial raster data type
- Topology data types
- US geocoder based on **TIGER** (short for **Topologically Integrated Geographic Encoding and Referencing**) census data
- A new JSONB data type, which allows indexing and the querying of JSON and GeoJSON

The PostGIS feature set is competitive among all geodatabases and is the most extensive among any open source or free geodatabases. The active momentum of the PostGIS development community is another reason for this system being best-of-breed. PostGIS is maintained at <http://postgis.net>.

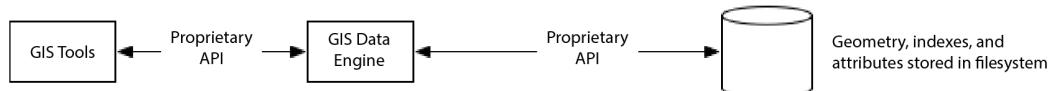
## Other spatially enabled databases

PostGIS is the gold standard among free and open source geospatial databases. However, there are several other systems that you should be aware of as a geospatial analyst. This list includes both commercial and open source systems, each with varying degrees of geospatial support.

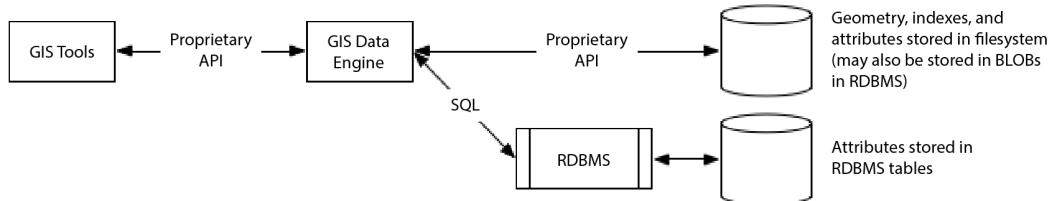
Geodatabases have evolved in parallel with geospatial software, standards, and the web. The internet has driven the need for large, multiuser geospatial database servers that are able to serve large amounts of data. The following diagram, courtesy of [www.OSGeo.org](http://www.OSGeo.org), shows how geospatial architectures have evolved, with a significant portion of this evolution happening at the database level:

## Evolution of GIS Architectures

### First-Generation GIS:



### Second-Generation GIS:



### Third-Generation GIS:

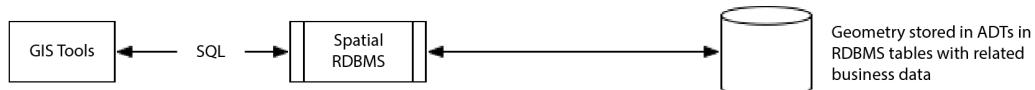


Figure 3.10 – Diagram showing the gradual evolution of spatial databases from completely proprietary APIs to more straightforward and open standards-based SQL queries

Now we'll step through some of the other widely used spatial databases to compare and contrast their similarities and differences.

### *Oracle Spatial and Graph*

The Oracle relational database is a widely used database system that is typically used by very large organizations because of its cost and large scalability. It is also extremely stable and fast. It runs some of the largest and most complicated databases in the world and is often found in hospitals, banks, and government agencies that manage millions of critical records.

The geospatial data capability first appeared in Oracle version 4 as a modification by the **Canadian Hydrographic Service (CHS)**. CHS also implemented Oracle's first spatial index, in the form of an unusual, but efficient, three-dimensional helical spiral. Oracle subsequently incorporated the modification and released the **Oracle Spatial Database Option (SDO)** in version 7 of the main database. The SDO system became Oracle Spatial in Oracle version 8. The database schema of Oracle Spatial still has the SDO prefix on some column and table names, similar to how PostGIS uses the OGC convention, ST, to separate spatial information from traditional relational database tables and functions at the schema level.

As of 2012, Oracle began calling the package **Oracle Spatial and Graph**, in order to emphasize the network data module. This module is used for analyzing networked datasets, such as transportation or utilities. However, the module can also be used against abstract networks, such as social networks. The analysis of social network data is a common target for big data analysis, which is now a growing trend. Big data social network analysis is likely the reason Oracle changed the name of the product.

Oracle Spatial has the following features:

- A geospatial data schema
- A spatial indexing system, which is now based on an R-tree index
- A SQL API for performing geometric operations
- A spatial data-tuning API to optimize a particular dataset
- A topology data model
- A network data model
- A GeoRaster data type to store, index, query, and retrieve raster data
- Three-dimensional data types, including **triangulated irregular networks (TINs)** and **LIDAR** (short for **Light Detection and Ranging**) point clouds
- A geocoder to search location names and return coordinates
- A routing engine for driving direction-type queries
- OGC compliance

Oracle Spatial and PostGIS are reasonably comparable and are both commonly used. You will see these two systems sooner or later as data sources when performing geospatial analysis.

#### Do you need Oracle Spatial and Graph?

Oracle Spatial and Graph is sold separately from Oracle itself. A little-known fact is that the SDO data type is native to the main Oracle database. If you have a simple application that just inputs points and retrieves them, you can use the main Oracle API to add, update, and retrieve SDO data without Oracle Spatial and Graph.

The US **Bureau of Ocean Energy, Management, Regulation, and Enforcement (BOEMRE)** uses Oracle to manage environmental, business, and geospatial data for billions of dollars, worth of oil, gas, and mineral rights, in one of the largest geospatial systems in the world. The following map is courtesy of BOEMRE:

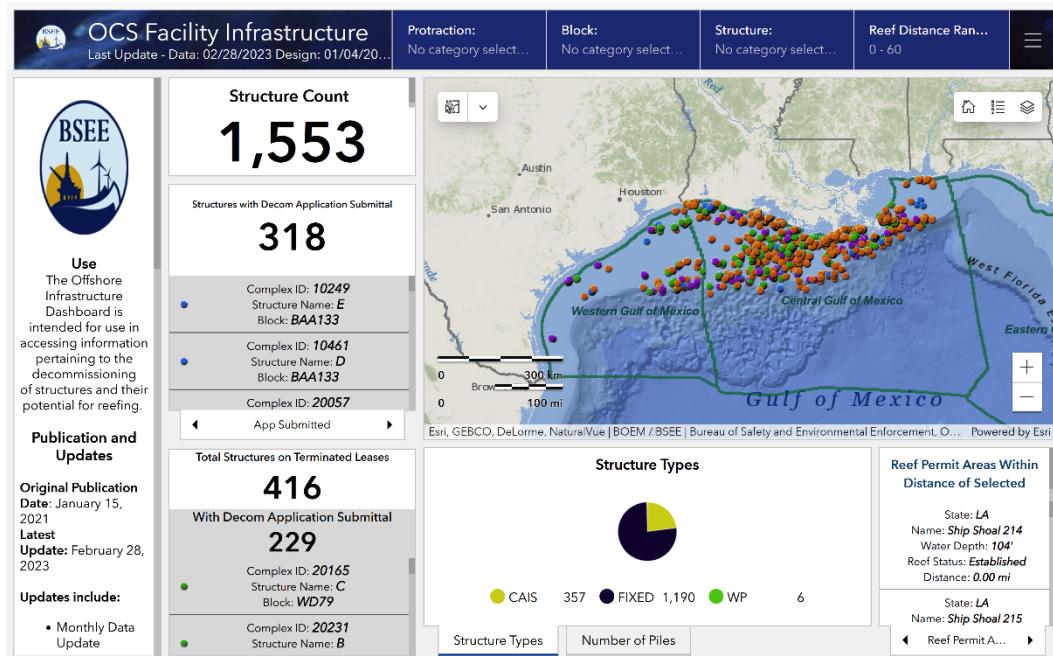


Figure 3.11 – The US Government website showing a spatial dashboard based on a massive Oracle Spatial infrastructure

### More about Oracle Spatial and Graph

Oracle Spatial and Graph can be found online at <http://www.oracle.com/us/products/database/options/spatial/overview>.

### ArcSDE

ArcSDE is Esri's **spatial data engine (SDE)**. It is now rolled into Esri's ArcGIS Server product after over a decade of being a standalone product. What makes ArcSDE interesting is that the engine is mostly database-independent, supporting multiple database backends. ArcSDE supports IBM DB2, Informix, Microsoft SQL Server, Oracle, and PostgreSQL as data storage systems. While ArcSDE has the ability to create and manage a spatial schema from scratch on systems such as Microsoft SQL Server and Oracle, it uses native spatial engines if they are available. This arrangement is the case for

IBM DB2, Oracle, and PostgreSQL. For Oracle, ArcSDE manages the table structure but can rely on the Oracle SDO data type for feature storage.

Like the previously mentioned geodatabases, ArcSDE also has a rich spatial selection API and can handle raster data. However, ArcSDE does not have as rich a SQL spatial API as Oracle and PostGIS. Esri technically supports basic SQL functionality related to ArcSDE, but it encourages users and developers to use Esri software or programming APIs in order to manipulate data that is stored through ArcSDE, as it is designed to be a data source for Esri software.

Esri does provide software libraries to allow developers to build applications outside of Esri software using ArcSDE or Esri's file-based geodatabase, known as a personal geodatabase. But these libraries are black boxes and the communication protocol that ArcSDE uses has never been reverse-engineered. Typically, interaction happens between ArcSDE and third-party applications at the web services level using the ArcGIS Server API (which supports OGC services to some degree), and a fairly straightforward REST API service that returns GeoJSON.

### ***Microsoft SQL Server***

Microsoft added spatial data support to its flagship database product in Microsoft SQL Server 2008. It has gradually improved since that version but still is nowhere near as sophisticated as Oracle Spatial or PostGIS. Microsoft supports the same data types as PostGIS, but uses slightly different naming conventions, with the exception of rasters, which are not directly supported. It also supports output to WKT and WKB formats.

It offers some very basic support for spatial selection, but it is obviously not a priority for Microsoft at the moment. This limited support is likely because it is all that can be used for Microsoft software mapping components, and several third-party engines can provide spatial support on top of SQL Server.

#### **More about Microsoft SQL Server**

Microsoft's support for spatial data in SQL Server is documented at <http://msdn.microsoft.com/en-us/library/bb933790.aspx>.

### ***MySQL***

MySQL, another highly popular and free database, provides nearly the same support as Microsoft SQL Server. The OGC geometry types are supported by basic spatial relationship functions. Through a series of buyouts, MySQL has become the property of Oracle.

While Oracle currently remains committed to MySQL as an open source database, this purchase has brought the ultimate future of the world's most popular open source database into question. But, as far as geospatial analysis is concerned, MySQL is barely a contender and is unlikely to be the first choice for any project.

**More about MySQL**

For more information on MySQL spatial support, visit <https://dev.mysql.com/doc/refman/8.0/en/spatial-types.html>.

***SpatiaLite***

SpatiaLite is an extension of the open source SQLite database engine. SQLite uses a file database and is designed to be integrated into applications rather than into the typical client-server model that is used by most relational database servers. SQLite already has spatial data types and spatial indexing, but SpatiaLite adds support for the OGC Simple Features specification, as well as map projections.

It should be noted that the extremely popular SQLite is not in the same category as Oracle, PostgreSQL, or MySQL, as it's a file-based database that is designed for single-user applications.

**More about SpatiaLite**

More on SpatiaLite can be found at <https://www.gaia-gis.it/fossil/libspatialite/index>.

***Esri File Geodatabase***

The Esri File Geodatabase is a file-based database format used to store geographic data, developed by Esri, and first introduced in 2004 as a replacement for the Personal Geodatabase format. The earlier Personal Geodatabase format had limitations in file size and user access. The new format was designed to be a scalable and flexible solution for managing large amounts of spatial data. It was also created to support multi-user editing, enabling collaboration on data projects. The Esri File Geodatabase was built using the Microsoft Access Database Engine, which provides a robust platform for storing and managing data. Over time, Esri has continued to enhance the File Geodatabase format by adding new features and improving performance. Today, the Esri File Geodatabase is an increasingly popular choice for managing and storing spatial data among GIS professionals and organizations around the world. It is an easy format for sharing data offline, however, it is just a data storage format and doesn't have a built-in relational database management system to manage complex data, so it is better for sharing relatively simple datasets.

**More about Esri's File Geodatabase**

You can read more about Esri's File Geodatabase format at <http://desktop.arcgis.com/en/arcmap/10.3/manage-data/administer-file-gdbs/file-geodatabases.htm>.

## GeoPackage

GeoPackage is a file-based geodatabase format. The official GeoPackage website, <http://geopackage.org>, describes it as follows:

*“An open, standards-based, platform-independent, portable, self-describing, compact format for transferring geospatial information.”*

It is also a direct answer to Esri's File Geodatabase format, and also the open geospatial community's designated shapefile killer, to replace the aging, partially-closed shapefile format. Both formats are really file specifications, which rely on other software in order to read and write data.

GeoPackage is an OGC specification, which means its future is secure as an industry data format. It is also a catch-all format, which can handle vector data, raster data, attribute information, and extensions to address new requirements. And, like any good database, it handles multiple layers. You can store an entire GIS project in a single package, therefore making data management much simpler.

## Routing

Routing is a very niche area of computational geometry. It is also a very rich field of study that goes far beyond the familiar driving directions use case. The requirements for a routing algorithm are simply a networked dataset and impedance values that affect the speed of travel on that network. Typically, the dataset is vector-based, but raster data can also be used for certain applications.

The two major contenders in this area are **Esri's Network Analyst** and the open source **pgRouting** engine for **PostGIS**. The most common routing problem is the most efficient way to visit a number of point locations. This problem is called the **Traveling Salesman Problem (TSP)**. The TSP is one of the most intensely studied problems in computational geometry. It is often considered the benchmark for any routing algorithm.

### More about the Traveling Salesman Problem

More information on the TSP can be found at [http://en.wikipedia.org/wiki/Travelling\\_salesman\\_problem](http://en.wikipedia.org/wiki/Travelling_salesman_problem).

## Esri Network Analyst and Spatial Analyst

Esri's entry into the routing arena, Network Analyst, is a truly generic routing engine that can tackle most routing applications regardless of context. Spatial Analyst is another Esri extension that is raster-focused, and it can perform least-cost path analysis on raster terrain data.

### More about ArcGIS Network Analyst

The ArcGIS Network Analyst product page is located on Esri's website at <http://www.esri.com/software/arcgis/extensions/networkanalyst>.

### **pgRouting**

The **pgRouting** extension for PostGIS adds routing functionality to the geodatabase. It is oriented toward road networks, but it can be adapted to work with other types of networked data.

The following diagram shows a driving distance radius calculation output by pgRouting, which is displayed in QGIS. The points are color-coded from green to red, based on their proximity to the starting location. As shown in the following diagram, the points are nodes in the network dataset, courtesy of QGIS.org (<https://qgis.org/en/site/>), which in this case are roads:

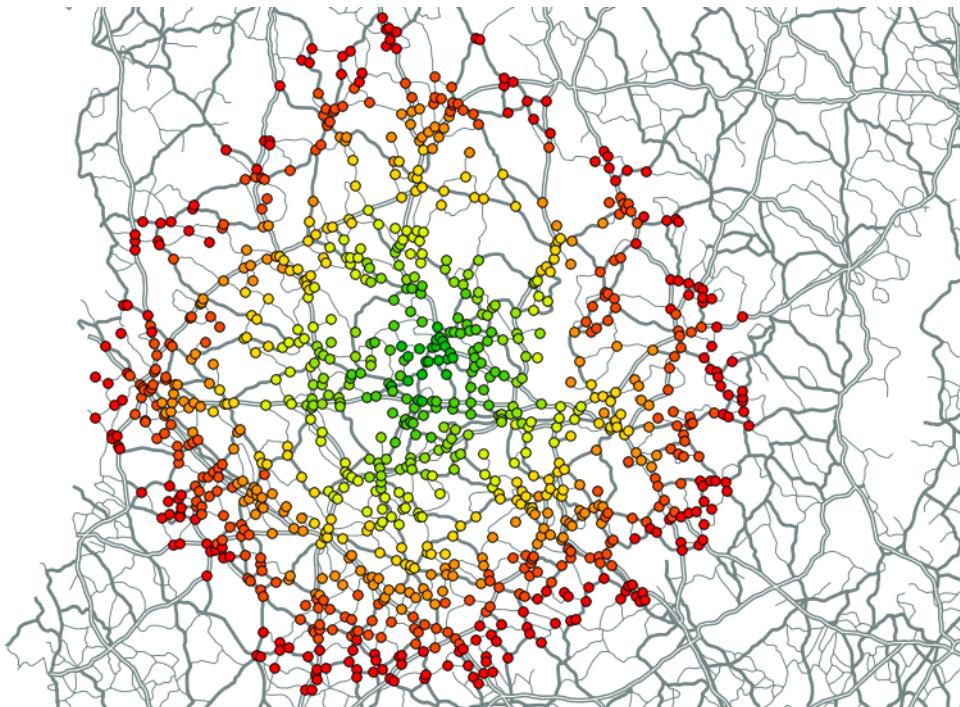


Figure 3.12 – Map in QGIS showing output from the pgRouting engines as a driving distance map showing different driving distances based on time with color-coded points placed on roads

### More about pgRouting

The pgRouting PostGIS extension is maintained at <http://pgrouting.org/>.

---

Next, we'll look at the tools that you need in order to visualize diagrams like the previous one.

## Understanding desktop tools (including visualization)

Geospatial analysis requires the ability to visualize output. This fact makes tools that can visualize data absolutely critical to the field. There are two categories of geospatial visualization tools.

The first is geospatial viewers and the second is geospatial analysis software. The first category – geospatial viewers—allows you to access, query, and visualize data, but not to edit it in any way. The second category allows you to perform those tasks, and edit the data, too. The main advantage of viewers is that they are typically lightweight pieces of software that launch and load data quickly.

Geospatial analysis software requires far more resources to be able to edit complex geospatial data, so it loads more slowly and often renders data more slowly, in order to provide dynamic editing functionality.

### Quantum GIS

**Quantum GIS**, more commonly known as **QGIS**, is a complete open source geographic information system. QGIS falls well within the geospatial analysis category in the two categories of visualization software. The development of the system began in 2002 and version 1.0 was released in 2009.

It is the best showcase of most of the libraries that have been previously mentioned in this chapter. QGIS is written in C++, using the Qt library for the GUI. The GUI is well-designed and easy to use. In fact, a geospatial analyst who has been trained on a proprietary package, such as Esri's ArcGIS or Manifold system, will be right at home using QGIS. The tools and menu system are logical and typical of a GIS system. The overall speed of QGIS is as good as, or better than, any other system that is available.

A nice feature of QGIS is that the underlying libraries and utility programs are just underneath the surface. Modules can be written by any third party in Python and added to the system. QGIS also has a robust online package management system to search for, install, and update these extensions. The Python integration includes a console that allows you to issue commands at the console and see the results in the GUI. QGIS isn't the only software to offer this capability.

Like most geospatial software packages, with Python integration, it installs a complete version of Python if you use the automated installer. There's no reason to worry if you already have Python installed. Having multiple versions of Python on a single machine is fairly common and well-supported. Many people have multiple versions of Python on their computers for the purpose of testing software, or because it is such a common scripting environment for so many different software packages.

When the Python console is running in QGIS, the entire program API is available through an automatically loaded object called `qgis.utils iface`. The following screenshot shows QGIS with the Python console running:

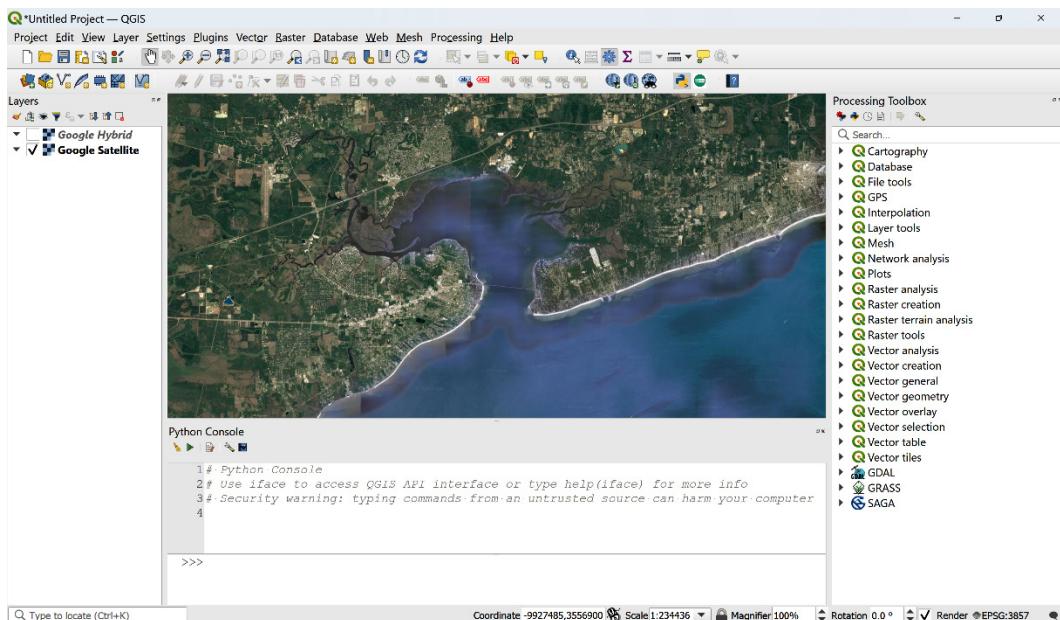


Figure 3.13 – QGIS screenshot showing the Python console at the bottom

Because QGIS is based on GDAL/OGR and GEOS, and it can use PostGIS, it supports all of the data sources that are offered by those packages. It also has nice raster processing features, too. QGIS works well for producing paper maps or entire map books using available extensions.

#### More about QGIS

QGIS is well documented on the QGIS website at <https://www.qgis.org/en/docs/index.html>. You can also find numerous online and video tutorials by searching for QGIS or a particular operation.

## GRASS GIS

GRASS is one of the oldest continuously developed geospatial systems in existence. The US Army Corps of Engineers began GRASS development in 1982. It was originally designed to run on Unix systems. In 1995, the army released the last patch, and the software was transferred to community development, where it has remained ever since.

Even though the user interface was redesigned, GRASS still feels somewhat esoteric to modern GIS users. However, because of its decades-old legacy and non-existent price tag, many geospatial workflows and highly specialized modules have been implemented in GRASS over the years, making it highly relevant to many organizations and individuals, especially in research communities. For these reasons, GRASS is still actively developed.

GRASS has also been integrated with QGIS, so the more modern and familiar QGIS GUI can be used to run GRASS functions. GRASS is also deeply integrated with Python and can be used as a library or as a command-line tool. The following screenshot shows some landform analysis in the native GRASS GUI, which was built using the wxPython library:

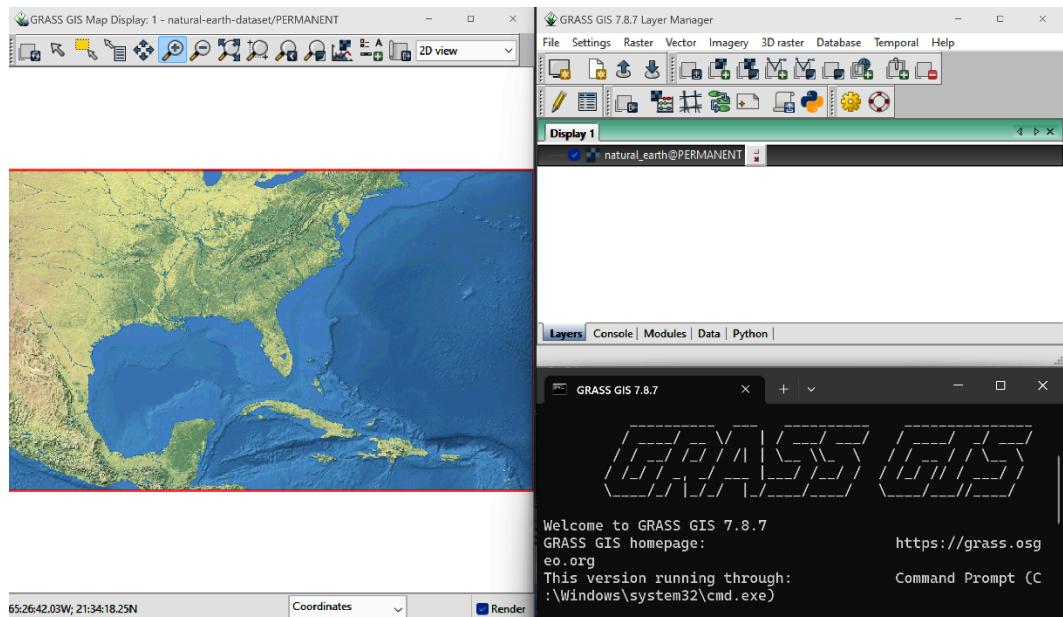


Figure 3.14 – Although GRASS is fully integrated into QGIS, the legacy GUI shown here still exists and can be used separately from QGIS

### More about GRASS

More about GRASS can be found online at <http://grass.osgeo.org/>.

## gvSIG

Another Java-based desktop GIS is **gvSIG**. The gvSIG project began in 2004 as part of a larger project to migrate the IT systems of the Regional Ministry of Infrastructure and Transport of Valencia, Spain to free software. The result was gvSIG, which has continued to mature. The feature set is mostly comparable to QGIS, with some unique capabilities as well.

The official gvSIG project has a very active fork called **gvSIG Community Edition (gvSIG CE)**. There is also a mobile version called gvSIG mobile. The gvSIG code base is open source.

### More about gvSIG

The official home page for gvSIG is available at <http://www.gvsig.org/web/>.

## OpenJUMP

**OpenJUMP** is another open source Java-based desktop GIS. JUMP stands for **Java Unified Mapping Platform** and was originally created by Vivid Solutions for the Government of British Columbia. After Vivid Solutions delivered JUMP, development stopped. Vivid Solutions eventually released JUMP to the open source community, where it was renamed OpenJUMP.

OpenJUMP has the ability to read and write shapefiles and OGC GML (short for **Geography Markup Language**) and supports PostGIS databases. It can also display some image formats and data from OGC WMS (short for **Web Map Server**) and WFS (short for **Web Feature Service**) services. It has a plugin architecture, and it can also serve as a development platform for custom applications.

### More about OpenJUMP

You can find out more about OpenJUMP on the official web page at <http://www.openjump.org/>.

## Google Earth

**Google Earth** is so ubiquitous that it hardly seems worth mentioning. The first release of EarthViewer 3D in 2001 (created by a company called Keyhole Inc.), and the EarthViewer 3D project were funded by the non-profit venture capital firm, In-Q-Tel, which in turn, is funded by the U.S. Central Intelligence Agency. This spy agency lineage and the subsequent purchase of Keyhole by Google to create and distribute Google Earth brought global attention to the geospatial analysis field.

Since the first release of the software as Google Earth in 2005, Google has continually refined it. Some of the notable additions are the creation of Google Moon, Google Mars, Google Sky, and Google Oceans. These are virtual globe applications, which feature data from the Moon and Mars, with the exception of Google Oceans, which adds sea-floor elevation mapping, known as bathymetry, to Google Earth.

Google Earth introduced the idea of the spinning virtual globe concept for the exploration of geographic data. After centuries of looking at 2D maps, or low-resolution physical globes, flying around Earth virtually and dropping onto a street corner anywhere in the world was mind-blowing – especially for geospatial analysts and other geography enthusiasts, as depicted in the following screenshot of Google Earth, overlooking the Central Business District in New Orleans, Louisiana:



Figure 3.15 – Google made 3D data and street-level virtual views widely available globally, which has advanced the types of analysis that can be done in GIS beyond two-dimensional maps

Just as Google revolutionized web mapping with its tile-based slippy mapping approach, the virtual globe concept was a major boost to geospatial visualization.

After the initial excitement wore off, many geospatial analysts realized that Google Earth was a very animated and fun geographic exploration tool, but it really had a very limited utility for any kind of meaningful geospatial analysis. Google Earth falls squarely into the realm of geospatial viewer software.

The only data format it consumes is its native **keyhole markup language (KML)**, which is an all-in-one data and styling format, which was discussed in *Chapter 2, Learning about Geospatial Data*. As this format is now an OGC standard, consuming only one data format immediately limits the utility of any tool. Any project involving Google Earth must first begin with complete data conversion and styling in KML, reminiscent of geospatial analysis from around 10-20 years ago. The tools that do support KML, including Google Maps, support a limited subset of KML.

Google Earth's native dataset has global coverage, but it is a mixture of datasets spanning several years and sources. Google has greatly improved the inline metadata in the tool, which identifies the source and approximate date of the current view. But this method creates confusion among lay people. Many people believe that the data in Google Earth is updated far more frequently than it really is. The Google Street View system, showing street-level, 360-degree views of much of the world, has somewhat helped to correct this misperception.

People are able to easily identify images of familiar locations as being several years old. Another common misperception created by Google Earth is that the entire world has been mapped in detail, and therefore creating a base map for geospatial analysis should be trivial. As discussed in *Chapter 2, Learning about Geospatial Data*, mapping an area of interest is far easier than it used to be a few years ago by using modern data and software, but it is still a complex and labor-intensive endeavor. This misperception is one of the first customer expectations a geospatial analyst must manage when starting a project.

Despite these misperceptions, the impact that Google has had on geospatial analysis is almost entirely positive. For decades, one of the most difficult challenges to growing the geospatial industry was convincing potential stakeholders that geospatial analysis is almost always the best approach when making decisions about people, resources, and the environment. This hurdle stands in sharp contrast to a car dealer. When a potential customer comes to a car lot, the salesman doesn't have to convince the buyer that they need a car, just about the type of car.

Geospatial analysts had to first educate project sponsors on the technology, and then convince them that the geospatial approach was the best way to address a challenge. Google has largely eliminated those steps for analysts.

#### More about Google Earth

Google Earth can be found online at <http://www.google.com/earth/index.html>.

## NASA WorldWind

**NASA WorldWind** is an open source virtual globe and geospatial viewer, originally released by the **US National Aeronautics and Space Administration (NASA)**, in 2004. It was originally based on Microsoft's .NET Framework, making it a Windows-centric application.

The following screenshot of NASA WorldWind looks similar to Google Earth:

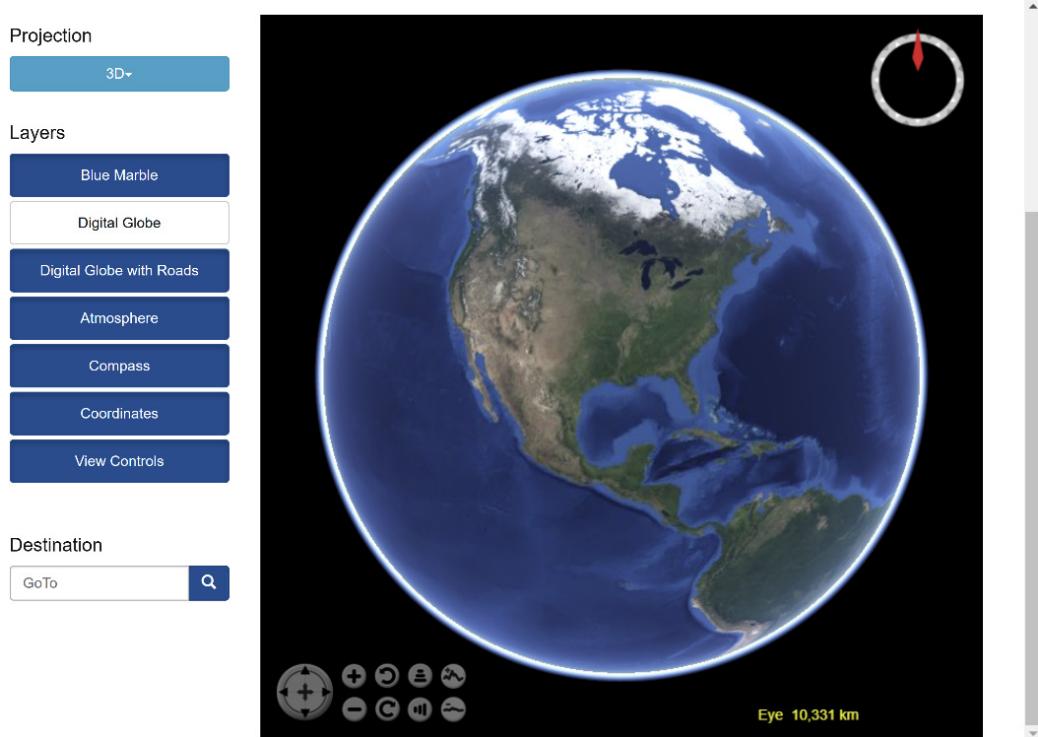


Figure 3.16 – A NASA WorldWind screenshot that looks strikingly similar to Google Earth but is built on an open source architecture

In 2007, a Java-based **software development kit (SDK)** was released, called WorldWind Java, which made WorldWind more cross-platform. The transition to Java also led to the creation of a browser plugin for WorldWind.

The WorldWind Java SDK is considered an SDK and not a desktop application like the .NET version. However, the demos included with the SDK provide a viewer without any additional development. While NASA WorldWind was originally inspired by Google Earth, its status as an open source project takes it in an entirely different direction.

Google Earth is a generalist tool that is bounded by the limits of the KML specification. NASA WorldWind is now a platform upon which anyone can develop without limits. As new types of data become available and computing resources grow, the potential of the virtual globe paradigm certainly holds more potential for geospatial visualization that has not yet been explored.

### More about NASA WorldWind

NASA WorldWind is available online at <http://worldwind.arc.nasa.gov/java/>.

## ArcGIS

Esri walks the line of being one of the greatest promoters of the geospatial analytical approach to understanding our world and is a privately held, profit-making business that must look out for its own interests to a certain degree. The ArcGIS software suite represents every type of geospatial visualization known, including vector, raster, globes, and 3D. It is also a market leader in many countries. As shown in the geospatial software map earlier in this chapter, Esri has increasingly incorporated open source software into its suite of tools, including GDAL for raster display, and Python as the scripting language for ArcGIS.

The following screenshot shows the core ArcGIS application, ArcMap from ArcGIS Pro, with satellite and topographic data analysis. The interface shares a lot in common with QGIS, as shown in this screenshot courtesy of the USGS:

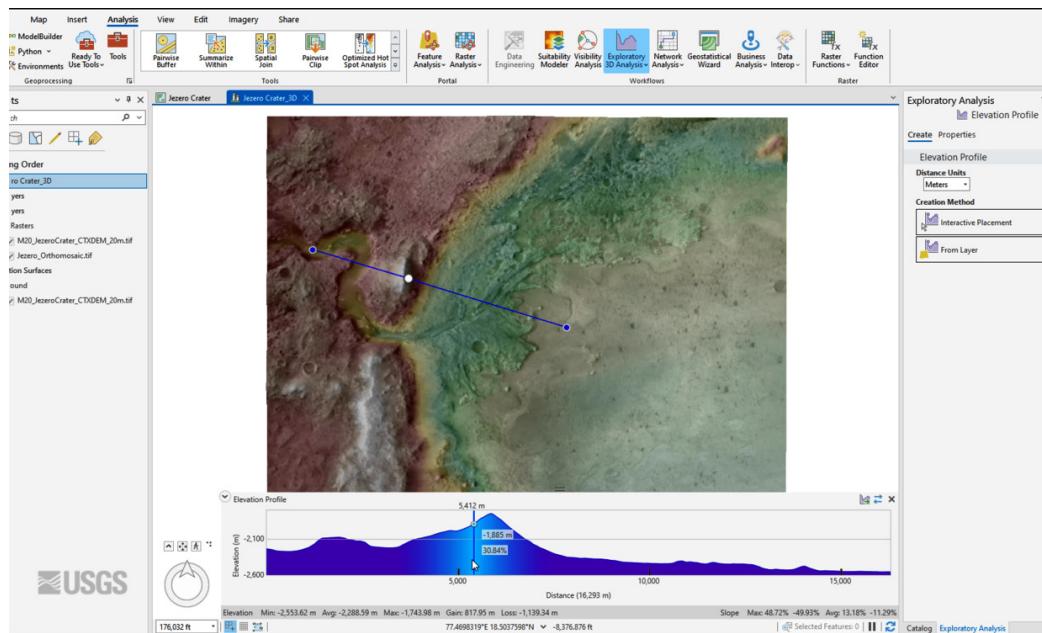


Figure 3.17 – An ArcGIS screenshot showing a typical desktop GIS application

### More about ArcGIS

The ArcGIS product page is available online at <http://www.esri.com/software/arcgis>.

## Leaflet and OpenLayers

Most of the screenshots in this chapter of web-based maps, as well as most of the online maps you encounter, are based on one of two JavaScript web mapping libraries: Leaflet or OpenLayers. These are not desktop tools but they are critical visualization tools in modern GIS because so much of mapping is now online.

Overall, these two libraries serve the same purpose of making it as easy as possible to create a map on a web page, which is why they're listed together here. But there are some key differences, as shown in the following table. Both libraries have their strengths and weaknesses for different applications and their performance ultimately comes down to the developer implementing them.

Category	Difference
Complexity	Leaflet has a simpler API and is generally easier to use, whereas OpenLayers can be more complex and may require more development experience
Features	OpenLayers has more features and supports a wider range of mapping providers and file formats, whereas Leaflet is more focused on providing a lightweight and flexible library
Popularity	Leaflet is more widely adopted in the web development community, and has a larger number of plugins and resources, whereas OpenLayers has a smaller community of users
Performance	Leaflet is generally faster and lighter, making it a better choice for mobile and low-bandwidth applications, whereas OpenLayers may use more resources and be slower to load
Licensing	Leaflet is released under the permissive MIT license, which allows commercial use without limitations, whereas OpenLayers is released under the more restrictive BSD license, which may require attribution and restrict commercial use

Table 3.2 – Table describing key differences between the Leaflet and OpenLayers JavaScript mapping libraries

### More about ArcGIS

The ArcGIS product page is available online at <http://www.esri.com/software/arcgis>.

Now that we understand the tools for visualizing and analyzing data, let's look at how to manage data.

## Understanding metadata management

The distribution of data on the internet has increased the importance of metadata. Data custodians are able to release a dataset to the entire world for download without any personal interaction. The metadata record of a geospatial dataset can follow this to help ensure that the integrity and accountability of this data are maintained.

Properly formatted metadata also allows for automated cataloging, search indexing, and the integration of datasets. Metadata has become so important that a common mantra within the geospatial community is data without metadata isn't data, meaning that a geospatial dataset cannot be fully utilized and understood without metadata.

The following sections list some of the commonly available metadata tools. The OGC standard for metadata management is the **Catalog Service for the Web (CSW)**, which creates a metadata-based catalog system and an API for distributing and discovering datasets.

### Python's pycsw library

`pycsw` is an OGC-compliant CSW for the publishing and discovery of geospatial metadata. It supports multiple APIs, including CSW 2 / CSW 3, OpenSearch, OAI-PMH, and SRU. It is extremely lightweight and pure Python. For an excellent example of a CSW and client that was built using `pycsw`, see the Pacific Islands **Ocean Observing System (PacIOOS)** catalog, available at <http://pacioos.org/search/>. The `pycsw` library is also used in a larger package called **GeoNode**.

### GeoNode

GeoNode is a Python-based geospatial content management system. It combines geospatial data creation, metadata, and visualization in a single server package. It also includes social features, such as comments and rating systems. It is open source and is available at <http://geonode.org/>. The following screenshot is from the GeoNode online demo:

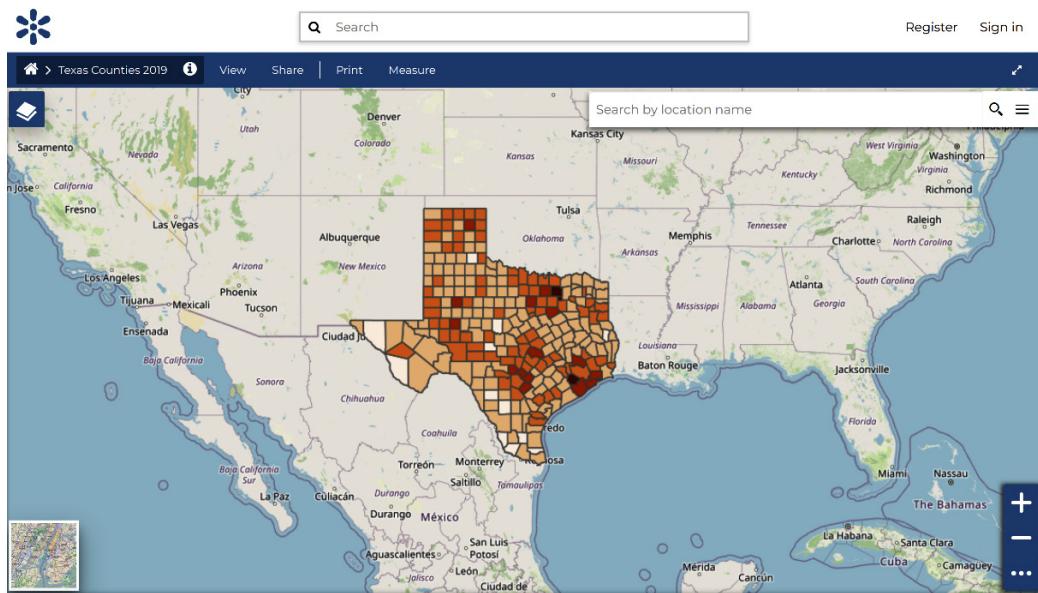


Figure 3.18 – This web map is an example of the GeoNode open source geospatial infrastructure, which provides a free alternative to the ArcGIS platform

GeoNode and pycsw are the two main metadata tools for Python. Next, we'll look at some tools that are written in other languages.

## GeoNetwork

**GeoNetwork** is an open source, Java-based catalog server used to manage geospatial data. It includes a metadata editor and search engine, as well as an interactive web map viewer. The system is designed to globally connect spatial data infrastructures. It can publish metadata through the web using metadata editing tools. It can also publish spatial data through the embedded GeoServer map server. It has user and group security permissions, and web and desktop configuration utilities.

GeoNetwork can also be configured in order to harvest metadata from other catalogs at scheduled intervals. The following screenshot is of the United Nations Food and Agriculture Organization's implementation of GeoNetwork:

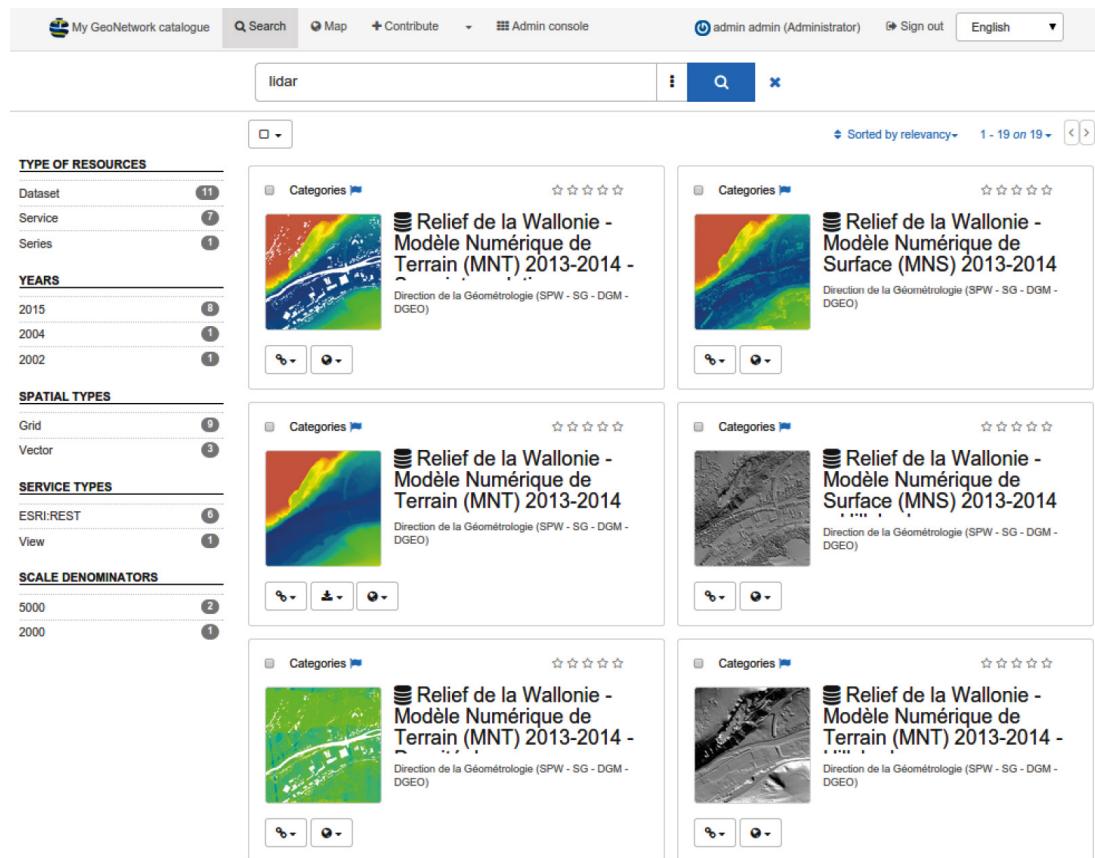


Figure 3.19 – This screenshot is a catalog view from the open source GeoNetwork platform

### More about GeoNetwork

You can find out more about GeoNetwork at <http://geonetwork-opensource.org/>.

## A quick look at artificial intelligence

We'll look at one more technology that doesn't fit in any other category and is starting to have an impact on the geospatial industry: **artificial intelligence** or AI. AI refers to a category of technology based on machine learning techniques and robust datasets to solve problems, which can include geospatial problems. One particular form of this technology that has received a lot of attention is **large language models (LLMs)**. An LLM is software trained on a very large body of text and uses statistical prediction to generate human-like text responses to questions or statements. The LLM that has received the most attention so far is called **ChatGPT**. ChatGPT can generate human-like narratives, but notably, it can also generate code. It was trained by reading a large portion of the internet, including lots of Python

code and geospatial knowledge. That means we can use ChatGPT to generate geospatial Python code almost automatically. There are limitations to ChatGPT's ability, but it is still an amazing tool for learning and performing geospatial analysis with Python. We'll learn more about how this works in an example in *Chapter 5, Python and Geospatial Algorithms*.

### More about ChatGPT

You can try out ChatGPT before we use it for geospatial analysis at <https://openai.com/blog/chatgpt>. Click on the **Try ChatGPT** button. You will have to create a login or use one of the available login methods. You will be presented with a chat interface where you can interactively converse with the system to ask it questions and shape its responses. There are several examples right in the interface to help you get started and begin exploring it.

## Summary

In this chapter, you learned about the hierarchy of geospatial analysis software. You also learned a framework for approaching the hundreds of existing geospatial software packages and libraries, by categorizing them into one or more major functions, including data access, computational geometry, raster processing, visualization, and metadata management.

We also examined commonly used foundation libraries, including GDAL, OGR, PROJ, and GEOS, which are found again and again in geospatial software. You can approach any new piece of geospatial software, trace it back to these core libraries, and then ask, “*What is the value added?*” to gain a better understanding of the package. If the software isn’t using one of these libraries, you need to ask, “*Why are these developers going against the grain? Does it address missing functionality or is it duplicating existing functionality?*” in order to understand what that system brings to the table.

Python was only mentioned a few times in this chapter so as to avoid any distraction in understanding the geospatial software landscape. But, as we will see, Python is interwoven into every single piece of software in this chapter, with the exception of JavaScript libraries, and is a fully capable geospatial tool in its own right. It is no coincidence that Python is the official scripting language of ArcGIS, QGIS, GRASS, and many other packages. It is also not by chance that GDAL, OGR, PROJ, CGAL, JTS, GEOS, and PostGIS all have Python bindings.

And as for the packages not mentioned here, they are all within Python’s grasp through the Jython Java distribution, the IronPython .NET distribution, Python’s various databases and web APIs, and the built-in `collections` module. As a geospatial analyst, if there’s one technology you can’t afford to pass up, it’s Python.

In the next chapter, we’ll see how Python comes into the picture in the geospatial industry. We’ll also learn about the GIS scripting language, the mashup glue language, and the full-blown programming language.

## Questions

1. What are the five general categories that geospatial software can be categorized into?
2. What two questions should you ask when you encounter new GIS software?

## Further reading

- <https://github.com/sacridini/Awesome-Geospatial>
- <https://github.com/jerr0328/awesome-geospatial-list>

# Part 2:

## Geospatial Analysis Concepts

This part represents the main building blocks of this book, where you'll be introduced to Python's role in the industry, with different code examples and data editing concepts. You'll learn about geospatial products and how they can be applied to solve problems. Then, you'll see how to practically work with remote sensing data using Python. At the end of this part, you'll learn how elevation data can be used in any geospatial format to analyze 3D features.

This part contains the following chapters:

- *Chapter 4, Geospatial Python Toolbox*
- *Chapter 5, Python and Geospatial Algorithms*
- *Chapter 6, Creating and Editing GIS Data*
- *Chapter 7, Python and Remote Sensing*
- *Chapter 8, Python and Elevation Data*



# 4

## Geospatial Python Toolbox

The first three chapters of this book covered the history of geospatial analysis, the types of geospatial data that are used by analysts, and the major software and libraries found within the geospatial industry. We used some simple Python examples here and there to illustrate certain points, but we mainly focused on the field of geospatial analysis, independent of any specific technology. Starting here, we will be using Python to conquer geospatial analysis and we will continue with that approach for the rest of this book. This chapter explains the software you will need in your toolbox to do just about anything you want in the geospatial field.

We'll discover the Python libraries that are used to access the different types of data that were found in the *Knowing about the most widely used vector data types* and *Understanding raster data types* sections of *Chapter 2, Learning about Geospatial Data*. Some of these libraries are pure Python, as well as some of the bindings to the different software packages that we looked at in *Chapter 3, The Geospatial Technology Landscape*.

In this chapter, we will cover the following topics:

- Using QGIS
- Installing third-party Python modules
- Jupyter
- Python networking libraries for acquiring data
- Bundling and compressing files
- Python markup and tag-based parsers
- **Well-Known Text (WKT)**
- Python JSON libraries
- OGR
- PyShp
- Shapely

- Fiona
- GDAL
- NumPy
- PIL
- PNGCanvas
- GeoPandas
- PyFPDF
- PyMySQL
- Rasterio
- OSMnx
- Folium

We will examine pure Python solutions whenever possible. Python is a very capable programming language, but some operations, particularly in remote sensing, are too computationally intensive and, therefore, are impractical when it comes to using pure Python or other interpreted languages. Fortunately, every aspect of geospatial analysis can be addressed in some way through Python, even if this is done by binding to a highly efficient C/C++/other compiled-language library.

We will avoid using broad scientific libraries that cover other domains beyond geospatial analysis to keep the solutions as simple as possible. There are many reasons to use Python for geospatial analysis, but one of the strongest arguments in its favor is its portability.

Furthermore, Python has been ported to Java as the Jython distribution and to the .NET **Common Language Runtime (CLR)** as IronPython. Python also comes in specialized versions such as Stackless Python for massively concurrent programs. There are versions of Python that are designed to run on cluster computers for distributed processing. Python is also available on many hosted application servers that do not allow you to install custom executables, such as the Google App Engine platform, which has a Python API.

By the end of this chapter, you will be familiar with the most important tools needed for geospatial analysis with Python.

## Technical requirements

You can find the code files for this chapter on GitHub at [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730\\_04\\_Asset\\_Files](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730_04_Asset_Files).

We will also be using the Anaconda environment, which is available at <https://www.anaconda.com/>.

## Using QGIS

Before we get started with creating geospatial data with Python, we'll need a way to view some of the datasets for the rest of the book. For that, we'll use the Python-friendly, open source QGIS. QGIS is a free and comprehensive geographic information system program. For this book, we will use it as a simple data viewer. You can download it for Windows, Linux, and macOS from <https://www.qgis.org>. Follow the instructions on the website to install it for your platform. Once installed, you will have a desktop icon to launch it. When you open QGIS, you will be presented with the **New empty project** option, which you can double-click to create a new map canvas. You can then drag files from your filesystem, both vector and raster, onto the map to view and interact with them. The following screenshot shows a satellite image with which we will work in this chapter, having been dragged from the filesystem onto the blank map canvas for viewing:

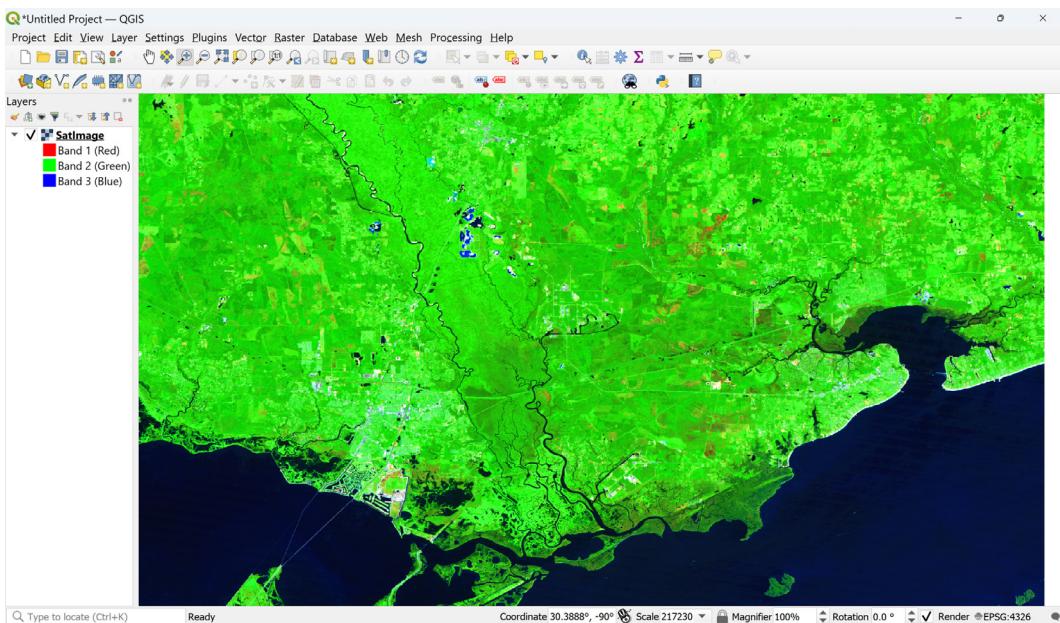


Figure 4.1 – QGIS allows you to quickly and easily view geospatial data by simply dragging it onto the map canvas

You can click and drag the map to pan around and use the magnifying glass tools to zoom in or out, or simply scroll your mouse wheel.

### The only GIS you'll ever need

Although we'll only be using QGIS for viewing data, you can use it for even the most advanced GIS and remote sensing analysis. To learn more about QGIS, check out *Learn QGIS – Fourth Edition* from Packt Publishing at <https://www.packtpub.com/product/learn-qgis-fourth-edition/9781788997423>.

Now that we have a way to look at the data we create, we can install some Python libraries to begin working with the data.

## Installing third-party Python modules

Modules written in pure Python (using the standard library) will mostly run on any of the 20 platforms mentioned on the Python website (<https://www.python.org/>). Each time you add a third-party module that relies on bindings to external libraries in other languages, you reduce Python's inherent portability. You also add a layer of complexity to fundamentally change the code by adding another language into the mix. Pure Python keeps things simple. Also, Python bindings to external libraries tend to be automatically or semi-automatically generated.

These automatically generated bindings are very generic and esoteric, and they simply connect Python to a C/C++ API using the method names from that API, instead of following the best practices for Python. There are, of course, notable exceptions to this approach that are driven by project requirements, which may include speed, unique library features, or frequently updated libraries where an automatically generated interface is preferable.

We'll make a distinction between modules that are included as a part of Python's standard library and modules that must be installed. In Python, the words *module* and *library* are used interchangeably. For most of this book, we'll use the powerful **Anaconda** environment. If a given library we need isn't available there, we'll fall back to Python's built-in installer pip.

## Anaconda

Python geospatial analysis requires that we use a variety of modules with many dependencies. These modules often build on each other using specific versions of the C or C++ libraries. You often run into version conflicts as you add Python modules to your system. Sometimes, when you upgrade a particular module, it might break your existing Python program due to changes in the API – or maybe you are running both Python 2 and Python 3 to take advantage of libraries written for each version. What you need is a way to safely install new modules without corrupting a working system or code. That's where Anaconda comes in.

Anaconda is an open source, cross-platform package management system that can also create and manage isolated environments called **virtual environments** to prevent frustrating library conflicts. The package manager program, **Conda**, within the Anaconda system makes it easy to install complex

packages, including geospatial ones. It also works with other languages besides Python, including R, Node.js, and Java.

Conda is very useful for setting up libraries and testing them so that we can try out new things in a development environment. It's usually better to custom-configure production environments, but Conda is a great way to prototype new ideas and, in the case of this book, learn how to perform geospatial analysis without becoming bogged down in installing software.

You can get started with Conda at <https://conda.io/en/latest/>. The documentation is available here: <https://docs.conda.io/en/latest/>.

Once Anaconda is installed, open a command prompt. On Windows, there is a special Anaconda PowerShell prompt in your Start menu. On other platforms, you can work from a standard terminal. The first thing we do is create an isolated environment for the scripts in this book that we'll call `lgawp4`. To create the environment, run the following command:

```
conda create -n lgawp4 python=3.10.9
```

This command tells Conda to create an environment named `lgawp4`, short for *Learning Geospatial Analysis – Fourth Edition*, and to install Python version `3.10.9` specifically so you'll be using the same version of Python used to create the scripts in the books to avoid any version differences.

Once Conda is done creating the environment, you can run the following:

```
conda activate lgawp4
```

This command enters you into the virtual environment, so any packages we install won't affect your system or other Anaconda environments. Once you're in the environment, you can install packages with the following command by replacing `<package_name>` with the library you want:

```
conda install <package_name>
```

## Miniconda

If you are on a system with limited resources, there is a version of Anaconda called **Miniconda** available on the Anaconda website that omits much of the larger packages in Anaconda and tries to create a more compact package management system.

Anaconda makes life easier for Python developers and includes another tool that does the same as well called Jupyter, but we won't be using this in our examples to keep things as simple as possible and avoid learning more software than we need to at once.

## Jupyter

The Jupyter project is something you should be aware of when working with geospatial or other scientific data. The Jupyter Notebook app creates and displays notebook documents in a web browser

as human-readable and machine-executable code and data. It's great for sharing tutorials for software and has become very common in the geospatial Python world.

You can find a good introduction to Jupyter Notebook and Python at [https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what\\_is\\_jupyter.html](https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html).

Now that we know how to install libraries and avoid software conflicts with the powerful Anaconda environment, let's look at Python's original package management system.

## PyPI and pip

PyPI acts as the official software repository for libraries and offers an easy-to-use setup program that simplifies the installation of packages, named pip, short for **Python Installation Program**. Installing a package is as simple as the following command:

```
pip install <package_name>
```

This command will search PyPI for a package matching that name. If the package isn't available on PyPI, you can point pip to the URL of a Python file, a ZIP file containing code, a Python package called a wheel, or even a file on your local filesystem. pip will do its best to manage dependencies but Anaconda is a much more powerful environment that makes it much easier to install some of the larger libraries we'll use when a lightweight, pure Python version isn't available.

Some very specialized libraries also don't have an Anaconda package available. In those cases, we'll fall back to pip as it is included in the Anaconda virtual environment. Both conda and pip work reasonably well together.

## The Python virtualenv module

Before Anaconda, developers used the Python built-in `virtualenv` module to create isolated virtual environments and prevent polluting their base Python installation with conflicting library versions. The `virtualenv` module is still included and may sometimes be used in more limited environments that don't support Anaconda.

### Getting started with virtualenv

To get started with `virtualenv`, follow the simple guide at <http://docs.python-guide.org/en/latest/dev/virtualenvs/>.

## Exploring Python virtualenv

The Python `virtualenv` module creates isolated, individual Python environments for each project so that you can avoid conflicting modules polluting your main Python installation. You can switch a particular environment on and off by activating or deactivating it. The `virtualenv` module is

efficient in that it doesn't actually copy your entire system's Python installation each time you create an environment. Let's get started:

1. Installing `virtualenv` is as simple as running the following code:

```
pip install virtualenv
```

2. Then, create a directory for your virtual Python environments. Name it whatever you want:

```
mkdir geospatial_projects
```

3. Now, you can create your first virtual environment using the following command:

```
virtualenv geospatial_projects/project1
```

4. Then, with the following command, you can activate the environment:

```
source geospatial_projects/project1/bin/activate
```

5. Now, when you run any Python commands in that directory, it will use the isolated virtual environment. When you're done, you can deactivate that environment with the following simple command:

```
deactivate
```

This is how you install, activate for use, and deactivate the `virtualenv` module.

Now that we know how to safely install Python packages in virtual environments, let's start using Python libraries to get things done! We'll start by learning how to download data with Python networking libraries.

## Python networking libraries for acquiring data

The vast majority of geospatial data sharing is accomplished via the internet, and Python is well equipped when it comes to networking libraries for almost any protocol. Automated data downloads are often an important step in automating a geospatial process. Data is typically retrieved from a website's **Uniform Resource Locator (URL)** or **File Transfer Protocol (FTP)** server and, because geospatial datasets often contain multiple files, the data is usually distributed as ZIP files.

A nice feature of Python is its concept of a file-like object. Most Python libraries that read and write data use a standard set of methods that allow you to access data from different types of resources, as if you were writing a simple file on disk. The networking modules in the Python standard library use this convention as well. The benefit of this approach is that it allows you to pass file-like objects to other libraries and methods that recognize this convention, avoiding a lot of setup that would otherwise be required for different types of data that are distributed in different ways.

## The Python `urllib` module

The Python `urllib` package is designed for simple access to any file with a URL address. The `urllib` package in Python 3 consists of several modules that handle different parts of managing web requests and responses. These modules implement some of Python's file-like object conventions, starting with its `open()` method. When you call `open()`, it prepares a connection to the resource but does not access any data. Sometimes, you just want to grab a file and save it to disk, instead of accessing it in memory. This function is available through the `urllib.request.retrieve()` method.

The following example uses the `urllib.request.retrieve()` method to download the zipped shapefile named `hancock.zip`, which is used in other examples. We define the URL and the local filename as variables. The URL is passed as an argument, along with the filename we want to use, to save it to our local machine, which, in this case, is just `hancock.zip`:

```
import urllib.request
url = "https://github.com/GeospatialPython/Learn/raw/master/hancock.
zip"
fileName = "hancock.zip"
out, response = urllib.request.urlretrieve(url, fileName)
print(f"successfully downloaded {out}")
print(response)
```

Then it will print the response:

```
successfully downloaded hancock.zip
Connection: close
Content-Length: 17341
Cache-Control: max-age=300
Content-Security-Policy: default-src 'none'; style-src 'unsafe-
inline'; sandbox
Content-Type: application/zip
ETag: "27b48bb2069d369cf9e3265594d6b9733ee524ee3298ad109e17773a2bf5387a"
Strict-Transport-Security: max-age=31536000
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
X-GitHub-Request-Id: A136:7701:10CBCB:14186F:64EB9BBB
Accept-Ranges: bytes
Date: Sun, 27 Aug 2023 18:54:55 GMT
Via: 1.1 varnish
X-Served-By: cache-dfw-kdfw8210118-DFW
X-Cache: HIT
X-Cache-Hits: 1
X-Timer: S1693162496.707037,VS0,VE2
```

```
Vary: Authorization,Accept-Encoding,Origin  
Access-Control-Allow-Origin: *  
Cross-Origin-Resource-Policy: cross-origin  
X-Fastly-Request-ID: 0af7e4ca038619b4e53ba57f1bc1729148a133f6  
Expires: Sun, 27 Aug 2023 18:59:55 GMT  
Source-Age: 67
```

The message from the underlying `httplib` module confirms that the file was downloaded to the current directory. The URL and filename could have been passed to the `retrieve()` method directly as strings as well. If you specify just the filename, the download saves to the current working directory. You can also specify a fully qualified pathname to save it somewhere else. You can also specify a callback function as a third argument, which will receive download status information for the file so that you can create a simple download status indicator or perform some other action.

The `urllib.request.urlopen()` method allows you to access an online resource with more precision and control. As we mentioned previously, it implements most of the Python file-like object methods with the exception of the `seek()` method, which allows you to jump to arbitrary locations within a file. You can read a file online one line at a time, read all the lines as a list, read a specified number of bytes, or iterate through each line of the file. All of these functions are performed in memory, so you don't have to store the data on disk. This ability is useful for accessing frequently updated data online that you may want to process without saving to disk.

In the following example, we demonstrate this concept by accessing the **United States Geological Survey (USGS)** earthquake feed to view all of the earthquakes in the world that have occurred within the last hour. This data is distributed as a **Comma-Separated Value (CSV)** file, which we can read line by line like a text file. CSV files are similar to spreadsheets and can be opened in a text editor or spreadsheet program:

1. First, you need to open the URL and read the header with the column names in the file.
2. Then, you need to read the first line, which contains a record of a recent earthquake, as shown in the following lines of code:

```
url = "http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_hour.csv"  
earthquakes = urllib.request.urlopen(url)  
earthquakes.readline() b'time,latitude,longitude,depth,mag,  
magType,nst,gap,dmin,rms,net,id,updated,place,type,  
horizontalError,depthError,magError,magNst,status,  
locationSource,magSource\n'  
earthquakes.readline()
```

This will produce an output listing for a recent earthquake incident similar to the following:

```
b'2023-08-27T19:01:07.110Z,32.2546667,-  
116.5703333,9.92,1.15,ml,11,253,0.09816,0.12,ci,ci39652562,2023-  
08-27T19:04:41.758Z,"33 km E of Primo Tapia, B.C.,  
MX",earthquake,0.73,0.85,0.198,6,automatic,ci,ci\n'
```

3. We can also iterate through this file – this is a memory-efficient way to read through large files.
4. If you are running this example in the Python interpreter, you will need to press the *Enter* or *Return* key twice to execute the loop. This action is necessary because it signals to the interpreter that you are done building the loop. In the following example, we abbreviate the output:

```
for record in earthquakes:  
    print(record)
```

This will list several earthquakes:

```
2013-06-14T14:30:40.000Z,62.0828,-145.2995,22.5,1.6,  
M1,8,108,0.08174669,0.86,ak,  
ak10739046,2013-06-14T14:37:02.318Z,"13km ESE of  
Glennallen, Alaska"  
...  
2013-06-14T13:42:46.300Z,38.8162,-122.8148,3.5,0.6, Md,,126,0.  
00898315,0.07,nc,nc72008115,2013-06-14T13:53:11.592Z,"6km NW of  
The Geysers, California"
```

## The Python requests module

The `urllib` module has been around for a long time. Another third-party module, `requests`, has been developed to make common HTTP requests even easier. The `requests` module has the following features:

- Keep-alive and connection pooling
- International domains and URLs
- Sessions with cookie persistence
- Browser-style SSL verification
- Automatic content decoding
- Basic/digest authentication
- Elegant key/value cookies
- Automatic decompression
- Unicode response bodies
- HTTP(S) proxy support
- Multipart file uploads
- Streaming downloads
- Connection timeouts

- Chunked requests
- .netrc support

In the following example, we'll download the same ZIP file we downloaded with the `urllib` module, except this time using the `requests` module. First, we need to install the `requests` module:

```
conda install requests
```

Then, we can import it:

```
import requests
```

Then, we can set up our variables for the URL and the output filename:

```
url = "https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/blob/main/B19730_04_Asset_Files/hancock.zip" fileName = "hancock.zip"
```

Retrieving the ZIP file is as simple as using the `requests` module's `get()` method:

```
r = requests.get(url)
```

Now, we can get the content from the `.zip` file and write it to our output file:

```
with open(fileName, 'wb') as f:  
    f.write(r.content)
```

The `requests` module has many more advanced features that are just as easy to use as this example. Now that we know how to get information via HTTP, let's examine another protocol, called FTP, that is often used to access geospatial data from online archives.

## FTP

FTP allows you to browse an online directory and download data using FTP client software. Until around 2004, when geospatial web services became very common, FTP was one of the most common ways to distribute geospatial data. FTP is less common now, but you occasionally encounter it when you're searching for data. Once again Python's "batteries-included" standard library has a reasonable FTP module called `ftplib` with a main class called `FTP()`.

In the following example, we will do the following:

1. We will access an FTP server hosted by the US **National Oceanic and Atmospheric Administration (NOAA)** to access a text file containing data from the organization's Tropical Atmospheric Ocean Buoy Network that's used to monitor our global climate. This particular buoy is off the coast of the western coast of Jakarta in the South Pacific.

2. We'll define the server and the directory path, and then we will access the server. All FTP servers require a username and password. Most public servers have a user called `anonymous` with the password also as `anonymous`, just like this one does.
3. Using Python's `ftplib` module, you can just call the `login()` method without any arguments to log in as the default anonymous user. Otherwise, you can add the username and password as string arguments.
4. Once we're logged in, we'll change to the directory containing the buoy data file.
5. To download the file, we'll open up a local file called `out` and pass its `write()` method as a callback function to the `ftplib.ftp.retrbinary()` method, which simultaneously downloads the file and writes it to our local file.
6. Once the file has been downloaded, we can close it to save it.
7. Then, we'll read the file and look for the line containing the ID, latitude, and longitude of the buoy to make sure that the data was downloaded successfully, as shown in the following lines of code:

```
# Read NOAA weather buoy location data via ftp
import ftplib
server = "ftp.pmel.noaa.gov"
dir = "taodata"
fileName = "taobuoypos.dat"
ftp = ftplib.FTP(server)
ftp.login()
ftp.cwd(dir)
with open(fileName, "wb") as out:
    ftp.retrbinary(f"RETR {fileName}", out.write)
with open(fileName) as tao:
    buoy = tao.readlines()[5]
    loc = buoy.split()
    print(f"Buoy {loc[0]} is located at {'\n'.join(loc[4:8])}")
ftp.quit()
```

The output is as follows:

```
Buoy DM467A is located at 8 03.1N 94 55.2W
```

In this example, we opened the local file in binary write mode, and we used the `retrbinary()` method from `ftplib`, as opposed to `retrlines()`, which uses ASCII mode. The binary mode works for both ASCII and binary files, so it's always a safer bet. In fact, in Python, the binary read and write modes for a file are only required on Windows.

If you are just downloading a simple file from an FTP server, note that many FTP servers have a web interface as well. In that case, you could use `urllib` to read the file. FTP URLs use the following format to access data:

```
ftp://username:password@server/directory/file
```

This format is insecure for password-protected directories because you are transmitting your login information over the internet. But for anonymous FTP servers, there is no additional security risk. To demonstrate this, the following example accesses the same file that we just saw but by using `urllib` instead of `ftplib`:

```
import urllib.request
tao = urllib.request.urlopen(
f"ftp://{{server}}/{{dir}}/{{fileName}}")
buoy = str(tao.readlines()[5], encoding="utf8")
loc = buoy.split()
print(f"Buoy {loc[0]} is located at {' '.join(loc[4:8])}")
```

Now that we can download files, let's learn how to decompress them.

## Bundling and compressing files

Geospatial datasets often consist of multiple files. For this reason, they are often distributed as ZIP or TAR file archives. These formats can also compress data, but their ability to bundle multiple files is the primary reason they are used for geospatial data. While the TAR format doesn't contain a compression algorithm, it incorporates `gzip` compression and offers it as a program option. Python has standard modules for reading and writing both ZIP and TAR archives. These modules are called `zipfile` and `tarfile`, respectively.

The following example extracts the `hancock.shp`, `hancock.shx`, and `hancock.dbf` files contained in the `hancock.zip` file we downloaded using `urllib` for use in the previous examples. This example assumes that the ZIP file is in the current directory:

```
import zipfile
zip = open("hancock.zip", "rb")
zipShape = zipfile.ZipFile(zip)
shpName, shxName, dbfName = zipShape.namelist()
shpFile = open(shpName, "wb")
shxFile = open(shxName, "wb")
dbfFile = open(dbfName, "wb")
shpFile.write(zipShape.read(shpName))
shxFile.write(zipShape.read(shxName))
dbfFile.write(zipShape.read(dbfName))
shpFile.close()
```

```
shxFile.close()  
dbfFile.close()
```

This example is more verbose than necessary for clarity. We can shorten this example and make it more robust by using a for loop around the `zipfile.namelist()` method without explicitly defining the different files as variables. This method is a more flexible and Pythonic approach, and could be used on ZIP archives with unknown contents, as shown in the following lines of code:

```
from zipfile import ZipFile  
zip = open("hancock.zip", "rb")  
zipShape = zipfile.ZipFile(zip)  
for fileName in zipShape.namelist():  
    ... out = open(fileName, "wb")  
    ... out.write(zipShape.read(fileName))  
    ... out.close()
```

Now that you understand the basics of the `zipfile` module, let's take the files we just unzipped and create a TAR archive with them. In this example, when we open the TAR archive for writing, we specify the write mode as `w:gz` for gzipped compression. We also specify the file extension as `.tar.gz` to reflect this mode, as shown in the following lines of code:

```
import tarfile  
tar = tarfile.open("hancock.tar.gz", "w:gz")  
tar.add("hancock.shp")  
tar.add("hancock.shx")  
tar.add("hancock.dbf")  
tar.close()
```

We can extract the files using the simple `tarfile.extractall()` method. First, we open the file using the `tarfile.open()` method and then extract it, as shown in the following lines of code:

```
tar = tarfile.open("hancock.tar.gz", "r:gz")  
tar.extractall()  
tar.close()
```

We'll work on one more example by combining elements we've learned in this chapter as well as the elements in the *Knowing about the most widely used vector data types* and *Understanding raster data types* sections of *Chapter 2, Learning about Geospatial Data*. We'll read the bounding box coordinates from the `hancock.zip` file without ever saving it to disk. We'll use the power of Python's file-like object convention to pass around the data.

Then, we'll use Python's `struct` module to read the bounding box, just as we did in *Chapter 2, Learning about Geospatial Data*.

In this case, we read the unzipped .shp file into a variable and access the data using Python array slicing by specifying the starting and ending indexes of the data separated by a colon (:). We are able to use list slicing because Python allows us to treat strings as lists. In this example, we also use Python's `StringIO` module to temporarily store data in memory in a file-like object that implements various methods, including the `seek()` method, which is absent from most Python networking modules, as shown in the following lines of code:

```
import urllib.request
import zipfile
import io
import struct
url = "https://github.com/PacktPublishing/Learning-Geospatial-
Analysis-with-Python-Fourth-Edition/blob/main/B19730_04_Asset_Files/
hancock.zip"
cloudshape = urllib.request.urlopen(url)
memoryshape = io.BytesIO(cloudshape.read())
zipshape = zipfile.ZipFile(memoryshape)
cloudshp = zipshape.read("hancock.shp")
# Access Python string as an array
struct.unpack("<dddd", cloudshp[36:68])
```

The result will be the bounding box of the shapefile:

```
(-89.6904544701547, 30.173943486533133, -89.32227546981174, 30.6483914
869749)
```

As you can see from the examples so far, Python's standard library packs a lot of punch. Most of the time, you don't have to download a third-party library just to access a file online.

## Python markup and tag-based parsers

Tag-based data, particularly different XML dialects, have become a very popular way to distribute geospatial data. Formats that are both machine- and human-readable are generally easy to work with, though they sacrifice storage efficiency for usability. These formats can become unmanageable for very large datasets but work very well in most cases.

While most formats are some form of XML (such as KML or GML), there is a notable exception. The **Well-Known Text (WKT)** format is fairly common but uses external markers and square brackets ([ ]) to surround data instead of tags in angled brackets around data like XML does.

Python has standard library support for XML, as well as some excellent third-party libraries available. Proper XML formats all follow the same structure, so you can use a generic XML library to read it. Because XML is text-based, it is often easy to write it as a string instead of using an XML library. The vast majority of applications that output XML do so in this way.

The primary advantage of using XML libraries for writing XML is that your output is usually validated. It is very easy to create an error while creating your own XML format. A single missing quotation mark can derail an XML parser and throw an error for somebody trying to read your data. When these errors happen, they virtually render your dataset useless. You will find that this problem is very common among XML-based geospatial data. What you'll discover is that some parsers are more forgiving of incorrect XML than others. Often, reliability is more important than speed or memory efficiency.

The analysis that's available at <http://lxml.de/performance.html> provides benchmarks for memory and speed among the different Python XML parsers.

## The minidom module

The Python `minidom` module is a very old and simple-to-use XML parser. It is part of Python's built-in set of XML tools in the `XML` package. It can parse XML files or XML that's been fed in as a string. The `minidom` module is best for small to medium-sized XML documents of less than about 20 MB before speed begins to decrease.

To demonstrate the `minidom` module, we'll use an example KML file, which is a part of Google's KML documentation that you can download. The data available at the following link represents timestamped point locations that have been transferred from a GPS device: <https://github.com/GeospatialPython/Learn/raw/master/time-stamp-point.kml>. Let's get started:

1. First, we'll parse this data by reading it in from the file and creating a `minidom` parser object. The file contains a series of `<Placemark>` tags, which hold a point and a timestamp at which that point was collected. So, we'll get a list of all of the placemarks in the file, and we can count them by checking the length of that list, as shown in the following lines of code:

```
from xml.dom import minidom
kml = minidom.parse("time-stamp-point.kml")
Placemarks = kml.getElementsByTagName("Placemark")
len(Placemarks) 361
```

2. As you can see, we retrieved all placemarks, which totaled 361. Now, let's take a look at the first `Placemark` element in the list:

```
Placemarks[0]
<DOM Element: Placemark at 0x2045a30>
```

3. Each `<Placemark>` tag is now a DOM element data type. To see what that element really is, we call the `toxml()` method, as follows:

```
Placemarks[0].toxml()
u'<Placemark>\n <TimeStamp>\n \n <when>2007-01-14T21:05:02Z</
when>\n
</TimeStamp>\n <styleUrl>#paddle-a</styleUrl>\n <Point>\n
```

```
<coordinates>-122.536226,37.86047,0</coordinates>\n</Point>\n </Placemark>'
```

4. The `toxml()` function outputs everything contained in the `Placemark` tag as a string object. If we wanted to print this information to a text file, we could call the `toprettyxml()` method, which would add additional indentation to make the XML more readable.
5. Now, what if we want to grab just the coordinates from this placemark? The coordinates are buried inside the `coordinates` tag, which is contained in the `Point` tag and nested inside the `Placemark` tag. Each element of a `minidom` object is called a node. Nested nodes are called children or child nodes. The child nodes include more than just tags – they can also include whitespace separating tags, as well as the data inside the tags. So, we can drill down to the `coordinates` tag using the tag name, but then we'll need to access the data node. All the `minidom` elements have `childNodes`, as well as a `firstChild()` method to access the first node.
6. We'll combine these methods to get to the data attribute of the first coordinate's data node, which we reference using index 0 in the list of `coordinates` tags:

```
coordinates = Placemarks[0].getElementsByTagName("coordinates")
point = coordinates[0].firstChild.data
print(point)
```

This will give us the coordinates of the point:

```
u'-122.536226,37.86047,0'
```

#### Why is the letter “u” in the results?

If you're new to Python, you'll notice that the text output in these examples is tagged with the letter u. This markup is how Python denotes Unicode strings that support internationalization to multiple languages with different character sets. Python 3.4.3 changes this convention slightly and leaves Unicode strings unmarked while marking UTF-8 strings with a b.

7. We can go a little further and convert this point string into usable data by splitting the string and converting the resulting strings into Python float types, as shown here:

```
x,y,z = point.split(", ")
x
u'-122.536226'
y u'37.86047'
z u'0'
x = float(x)
y = float(y)
z = float(z)
print(x,y,z)
```

The preceding code will give us the *x*, *y*, and *z* coordinates as follows:

```
(-122.536226, 37.86047, 0.0)
```

8. Using Python list comprehension, we can perform this operation in a single step, as you can see in the following lines of code:

```
x,y,z = [float(c) for c in point.split(",")]
print(x,y,z)
```

And once again we get the coordinates:

```
(-122.536226, 37.86047, 0.0)
```

This example scratches the surface of what the `minidom` library can do. For a great tutorial on this library, have a look at the tutorial at <https://www.edureka.co/blog/python-xml-parser-tutorial/>.

## The ElementTree module

The `minidom` module is pure Python, easy to work with, and has been around since Python 2.0.

However, Python 2.5 added a more efficient yet high-level XML parser to the standard library, called `ElementTree`. `ElementTree` is interesting because it has been implemented in multiple versions.

There is a pure Python version and a faster version written in C called `cElementTree`. You should use `cElementTree` wherever possible, but it's possible that you may find yourself on a platform that doesn't include the C-based version. When you import `cElementTree`, you do a quick test to see whether it's available and then fall back to the pure Python version if necessary:

```
try:
    import xml.etree.cElementTree as ET
except ImportError:
    import xml.etree.ElementTree as ET
```

One of the great features of `ElementTree` is its implementation of a subset of the XPath query language. XPath is short for XML Path and allows you to search an XML document using a path-style syntax. If you work with XML frequently, learning XPath is essential. You can learn more about XPath at [https://www.w3schools.com/xml/xpath\\_intro.asp](https://www.w3schools.com/xml/xpath_intro.asp).

One catch with this feature is that if the document specifies a namespace, as most XML documents do, you must insert that namespace into queries. `ElementTree` does not automatically handle the namespace for you. Your options are to manually specify it or try to extract it using string parsing from the root element's tag name.

We'll repeat the `minidom` XML parsing example using `ElementTree`:

1. First, we'll parse the document and then we'll manually define the KML namespace; later, we'll use an XPath expression and the `find()` method to find the first `Placemark` element.
2. Finally, we'll find the coordinates and the child node and then grab the text containing the latitude and longitude.

In both cases, we could have searched directly for the coordinates tag. But, by grabbing the `Placemark` element, it gives us the option of grabbing the corresponding timestamp child element later, if we so choose, as shown in the following lines of code:

```
tree = ET.ElementTree(file="time-stamp-point.kml")
ns = "{http://www.opengis.net/kml/2.2}"
placemark = tree.find("./%sPlacemark" % ns)
coordinates = placemark.find("./{}Point/{}/{}coordinates".format(ns, ns))
print(coordinates.text)
```

And we'll see the coordinates as follows:

```
'-122.536226,37.86047,0'
```

In this example, notice that we used the Python string formatting syntax, which is based on the string formatting concept found in C. When we defined the XPath expression for the `placemark` variable, we used the `%` placeholder to specify the insertion of a string. Then, after the string, we used the `%` operator followed by a variable name to insert the `ns` namespace variable where the placeholder is. In the `coordinates` variable, we used the `ns` variable twice, so we specified a tuple containing `ns` twice after the string.

#### Learn more about string formatting

String formatting is a simple yet extremely powerful and useful tool in Python that's worth learning. You can find more information in Python's documentation online at <https://docs.python.org/3/library/string.html>.

## Building XML using `ElementTree` and `minidom`

Most of the time, XML can be built by concatenating strings, as you can see in the following code:

```
xml = "<?xml version='1.0' encoding='utf-8'?>"
xml += "<kml xmlns='http://www.opengis.net/kml/2.2'>" xml += "
<Placemark>"
xml += " <name>Office</name>"
xml += " <description>Office Building</description>" xml += " <Point>"
```

```

xml += " <coordinates>"
xml += " -122.087461,37.422069"
xml += " </coordinates>" xml += " </Point>"
xml += " </Placemark>" xml += "</kml>"
```

However, this method can be quite prone to typos, which creates invalid XML documents. A safer way is to use an XML library. Let's build this simple KML document using `ElementTree`:

1. We'll define the `rootKML` element and assign it a namespace.
2. Then, we'll systematically append sub-elements to the root, wrap the elements as an `ElementTree` object, declare the XML encoding, and write it out to a file called `placemark.xml`, as shown in the following lines of code:

```

root = ET.Element("kml")
root.attrib["xmlns"] = "http://www.opengis.net/kml/2.2"
placemark = ET.SubElement(root, "Placemark")
office = ET.SubElement(placemark, "name")
office.text = "Office"
point = ET.SubElement(placemark, "Point")
coordinates = ET.SubElement(point, "coordinates")
coordinates.text = "-122.087461,37.422069, 37.422069"
tree = ET.ElementTree(root)
tree.write("placemark.kml", xml_declaration=True, encoding='utf-8', method="xml")
```

The output is identical to the previous string building example, except that `ElementTree` does not indent the tags but rather writes it as one long string. The `minidom` module has a similar interface, which is documented in the book *Dive Into Python*, by Mark Pilgrim, which was referenced in the `minidom` example that we just saw.

### Dive Into Python

The excellent books *Dive Into Python* and *Dive Into Python 3* (<https://diveintopython3.net/>), both by Mark Pilgrim, are excellent Python references. Both are mentioned because the original book has an excellent section on the `minidom` module, which is replaced by `ElementTree` in the new book.

XML parsers such as `minidom` and `ElementTree` work very well on perfectly formatted XML documents. Unfortunately, the vast majority of XML documents out there don't follow these rules and contain formatting errors or invalid characters. You'll find that you are often forced to work with this data and must resort to extraordinary string parsing techniques to get the small subset of data you actually need. But thanks to Python and Beautiful Soup, you can elegantly work with bad and even terrible tag-based data.

Beautiful Soup is a module that was specifically designed to robustly handle broken XML. It is oriented toward HTML, which is notorious for incorrect formatting but works with other XML dialects too. Beautiful Soup can work with several XML parsers but defaults to `lxml` if it's available, so we'll install that too. Beautiful Soup and `lxml` can be installed with the following commands:

```
conda install beautifulsoup4
conda install lxml
```

Then, to use it, you simply import it as follows:

```
from bs4 import BeautifulSoup
```

To try it out, we'll use a **GPS Exchange Format (GPX)** tracking file from a smartphone application, which has a glitch and exports slightly broken data. You can download this example file from [https://raw.githubusercontent.com/GeospatialPython/Learn/master/broken\\_data.gpx](https://raw.githubusercontent.com/GeospatialPython/Learn/master/broken_data.gpx).

This 2,347-line data file is in pristine condition except that it is missing a closing `</trkseg>` tag, which should be located at the very end of the file, just before the closing `</trk>` tag. This error was caused by a data export function in the source program. This defect is most likely a result of the original developer manually generating the GPX XML on export and forgetting the line of code that adds this closing tag. Watch what happens if we try to parse this file with `minidom`:

```
gpx = minidom.parse("broken_data.gpx") Traceback (most recent call
last):
File "<stdin>", line 1, in <module>
File "C:\Python34\lib\xml\dom\minidom.py", line 1914, in parse
return expatbuilder.parse(file)
File "C:\Python34\lib\xml\dom\expatbuilder.py", line 924, in parse
result = builder.parseFile(fp)
File "C:\Python34\lib\xml\dom\expatbuilder.py", line 207, in parseFile
parser.Parse(buffer, 0)
xml.parsers.expat.ExpatError: mismatched tag: line 2346, column 2
```

As you can see from the last line in the error message, the underlying XML parser in `minidom` knows exactly what the problem is – a mismatched tag right at the end of the file. However, it refused to do anything more than report the error. You must have perfectly formed XML or none at all to avoid this.

Now, let's try the more sophisticated and efficient `ElementTree` module with the same data:

```
ET.ElementTree(file="broken_data.gpx") Traceback (most recent call
last):
File "<stdin>", line 1, in <module>
File "C:\Python34\lib\xml\etree\ElementTree.py", line 611, in
    init self.parse(file)
File "C:\Python34\lib\xml\etree\ElementTree.py", line 653, in parse
    parser.feed(data)
```

```
File "C:\Python34\lib\xml\etree\ElementTree.py", line 1624, in feed
    self._raiseerror(v)
File "C:\Python34\lib\xml\etree\ElementTree.py", line 1488, in
    _raiseerror raise err
xml.etree.ElementTree.ParseError: mismatched tag: line 2346, column 2
```

As you can see, different parsers face the same problem. Poorly formed XML is an all too common reality in geospatial analysis, and every XML parser assumes that all the XML in the world is perfect, except for one. Enter Beautiful Soup. This library shreds bad XML into usable data without a second thought, and it can handle far worse defects than missing tags. It will work despite missing punctuation or other syntax and will give you the best data it can. It was originally developed for parsing HTML, which is notoriously horrible for being poorly formed, but it works fairly well with XML as well, as shown here:

```
from bs4 import BeautifulSoup
gpx = open("broken_data.gpx")
soup = BeautifulSoup(gpx.read(), features="xml")
```

No complaints from Beautiful Soup! Just to make sure the data is actually usable, let's try and access some of the data. One of the fantastic features of Beautiful Soup is that it turns tags into attributes of the parse tree. If there are multiple tags with the same name, it grabs the first one. Our example data file has hundreds of `<trkpt>` tags. Let's access the first one:

```
soup.trkpt
<trkpt lat="30.307267000" lon="-89.332444000">
<ele>10.7</ele><time>2013-05-16T04:39:46Z</time></trkpt>
```

We're now certain that the data has been parsed correctly and that we can access it. If we want to access all of the `<trkpt>` tags, we can use the `findAll()` method to grab them and then use the built-in Python `len()` function to count them, as shown here:

```
tracks = soup.findAll("trkpt")
len(tracks)
2321
```

If we write the parsed data back out to a file, Beautiful Soup outputs the corrected version. We'll save the fixed data as a new GPX file using the Beautiful Soup module's `prettyify()` method to format the XML with nice indentation, as you can see in the following lines of code:

```
fixed = open("fixed_data.gpx", "w")
fixed.write(soup.prettyify())
fixed.close()
```

Beautiful Soup is a very rich library with many more features. To explore it further, visit the Beautiful Soup documentation online at <http://www.crummy.com/software/BeautifulSoup/bs4/documentation.html>.

## Well-Known Text (WKT)

The WKT format has been around for years and is a simple text-based format for representing geometries and spatial reference systems. It is primarily used as a data exchange format by systems that implement the OGC Simple Features for SQL specification. Take a look at the following example WKT representation of a polygon:

```
POLYGON((0 0, 4 0, 4 4, 0 4, 0 0), (1 1, 2 1, 2 2, 1 2, 1 1))
```

Currently, the best way to read and write WKT is by using the `Shapely` library. `Shapely` provides a very Python-oriented or Pythonic interface to the **Geometry Engine, Open Source (GEOS)** library we described in *Chapter 3, The Geospatial Technology Landscape*.

You can install `Shapely` using Conda in your virtual environment. You can also use the wheel from the site we mentioned in the previous section. `Shapely` has a WKT module that can load and export this data. Let's use `Shapely` to load the previous polygon example and then verify that it has been loaded as a polygon object by calculating its area:

```
import shapely.wkt
wktPoly = "POLYGON((0 0, 4 0, 4 4, 0 4, 0 0), (1 1, 2 1, 2 2, 1 2,
1 1))"
poly = shapely.wkt.loads(wktPoly)
poly.area 15.0
```

We can convert any `Shapely` geometry back into WKT by simply calling its `wkt` attribute, as shown here:

```
print(poly.wkt)
```

This will give us the WKT representation of the polygon:

```
'POLYGON ((0.0 0.0, 4.0 0.0, 4.0 4.0, 0.0 4.0, 0.0 0.0), (1.0 1.0,
2.0 1.0, 2.0 2.0, 1.0 2.0, 1.0 1.0))'
```

`Shapely` can also handle the WKT binary counterpart called **Well-Known Binary (WKB)**, which is used to store WKT strings as binary objects in databases. `Shapely` loads WKB using its `wkb` module in the same way as the `wkt` module, and it can convert geometries by calling that object's `wkb` attribute.

`Shapely` is the most Pythonic way to work with WKT data, but you can also use the Python bindings to the OGR library, which we installed earlier in this chapter as part of GDAL.

For this example, we'll use a shapefile with one simple polygon, which can be downloaded as a ZIP file. It is available at <https://github.com/GeospatialPython/Learn/raw/master/polygon.zip>.

In the following example, we'll open the `polygon.shp` file from the shapefile dataset, call the required `GetLayer()` method, get the first (and only) feature, and then export it to WKT. We'll be using the OGR facility that we installed as part of GDAL earlier in this chapter:

```
from osgeo import ogr
shape = ogr.Open("polygon.shp")
layer = shape.GetLayer()
```

Now we'll iterate over the available feature:

```
feature = layer.GetNextFeature()
```

This next line will return the feature geometry or `None`, which could be used as a validity check if needed:

```
geom = feature.GetGeometryRef()
wkt = geom.ExportToWkt()
print(wkt)
```

The preceding will once again give us a WKT string:

```
' POLYGON ((-99.904679362176353 51.698147686745074,
-75.010398603076666 46.56036851832075,-75.010398603076666
46.56036851832075,-75.010398603076666 46.56036851832075,
-76.975736557742451 23.246272688996914,-76.975736557742451
23.246272688996914,-76.975736557742451 23.246272688996914,
-114.31715769639194 26.220870210283724,-114.31715769639194
26.220870210283724,-99.904679362176353 51.698147686745074))'
```

Note that with OGR, you have read access to each feature and export it individually, since the `ExportToWkt()` method is at the feature level. We can now turn around and read a WKT string using the `wkt` variable containing the export. We'll import it back into `ogr` and get the bounding box, also known as an envelope, of the polygon, as you can see here:

```
poly = ogr.CreateGeometryFromWkt(wkt)
poly.GetEnvelope()
```

And we'll get the envelope coordinates:

```
(-114.31715769639194, -75.01039860307667, 23.246272688996914,
51.698147686745074)
```

Shapely and OGR are used for reading and writing valid WKT strings. Of course, just like XML, which is also text, you could manipulate small amounts of WKT as strings in a pinch. Next, we'll look at a modern text format that is becoming very common in the geospatial world.

## Python JSON libraries

**JavaScript Object Notation (JSON)** is rapidly becoming the number-one data exchange format across a lot of fields. Its lightweight syntax and its similarity to existing data structures in both the JavaScript that Python borrows some data structures from and JavaScript itself make it a perfect match for Python.

The following GeoJSON example document contains a single point:

```
{  
    "type": "Feature",  
    "id": "OpenLayers.Feature.Vector_314",  
    "properties": {},  
    "geometry": {  
        "type": "Point",  
        "coordinates": [  
            97.03125,  
            39.7265625  
        ]  
    },  
    "crs": {  
        "type": "name",  
        "properties": {  
            "name": "urn:ogc:def:crs:OGC:1.3:CRS84"  
        }  
    }  
}
```

This example is just a simple point with new attributes that would be stored in the properties data structure of the geometry. The ID, coordinates, and CRS information in the preceding example would change depending on your particular dataset.

Let's modify this example GeoJSON document using Python. First, we'll compact the example document into a single string to make it easier to handle:

```
jsdata = """{ "type": "Feature", "id": "OpenLayers.Feature.Vector_314",  
"properties": {},  
"geometry": {  
    "type": "Point",  
    "coordinates": [  
        97.03125,
```

```
    39.7265625
]
},
"crs": {
    "type": "name",
    "properties": {
        "name": "urn:ogc:def:crs:OGC:1.3:CRS84"
    }
}
}"""
}
```

Now, we can use the GeoJSON `jsdata` string variable, created in the preceding code, in the following examples.

## The json module

GeoJSON looks very similar to a nested set of Python's dictionaries and lists. Just for fun, let's just try and use Python's `eval()` function to parse it as Python code:

```
point = eval(jsdata)
print(point["geometry"])
```

This will give us the GeoJSON for the point:

```
{'type': 'Point', 'coordinates': [97.03125, 39.7265625]}
```

Wow! That worked! We turned that random GeoJSON string into native Python data in one easy step. Keep in mind that the JSON data format is based on JavaScript syntax, which happens to be similar to Python. Also, as you get deeper into GeoJSON data and work with larger data, you'll find that JSON allows characters that Python does not. Using Python's `eval()` function is considered very insecure as well. But as far as keeping things simple is concerned, note that it doesn't get any simpler than that!

Thanks to Python's drive toward simplicity, the more advanced method doesn't get much more complicated. Let's use Python's `json` module, which is part of the standard library, to turn the same string into Python the right way:

```
import json
json.loads(jsdata)
```

We'll see the following GeoJSON:

```
{u'geometry': {u'type': u'Point', u'coordinates': [97.03125,
39.7265625]}, u'crs': {u'type': u'name', u'properties': {u'name':
'urn:ogc:def:crs:OGC:1.3:CRS84'}}, u'type': u'Feature', u'id':
u'OpenLayers.Feature.Vector_314',
```

```
u'properties':  
{}}
```

As a side note, in the previous example, the CRS84 property is a synonym for the common WGS84 coordinate system. The `json` module adds some nice features such as safer parsing and conversion of strings into Unicode. We can export Python data structures to JSON in almost the same way:

```
pydata = json.loads(jsdata)  
json.dumps(pydata)
```

This will output the following GeoJSON:

```
{ "geometry": { "type": "Point", "coordinates": [97.03125,  
39.7265625]}, "crs": { "type": "name", "properties": { "name":  
"urn:ogc:def:crs:OGC:1.3:CRS84" }}, "type": "Feature", "id":  
"OpenLayers.Feature.Vector_314", "properties":  
{ } }
```

When you dump data, it comes out as one long string that's difficult to read. There's a way we can print the data so it is easier to read – by passing the `dumps()` method an `indent` value:

```
print(json.dumps(pydata, indent=4))
```

And we'll see the following GeoJSON:

```
{  
    "type": "Feature",  
    "id": "OpenLayers.Feature.Vector_314",  
    "properties": {},  
    "geometry": {  
        "type": "Point",  
        "coordinates": [  
            97.03125,  
            39.7265625  
        ]  
    },  
    "crs": {  
        "type": "name",  
        "properties": {  
            "name": "urn:ogc:def:crs:OGC:1.3:CRS84"  
        }  
    }  
}
```

Now that we understand the `json` module, let's look at the geospatial version called `geojson`. You can install the `geojson` module as follows:

```
pip install geojson
```

Now let's learn how to use it.

## The `geojson` module

We could happily go on forever reading and writing GeoJSON data using the `json` module, but there's an even better way. The `geojson` module that's available on PyPI offers some distinct advantages. For starters, it knows the requirements of the GeoJSON specification, which can save a lot of typing. Let's create a simple point using this module and export it to GeoJSON:

```
import geojson
p = geojson.Point([-92, 37])
```

This time, when we dump the JSON data for viewing, we'll add an `indent` argument with a value of 4 so that we get nicely indented JSON data that's easier to read:

```
geojs = geojson.dumps(p, indent=4)
geojs
```

Our output is as follows:

```
{
    "type": "Point",
    "coordinates": [
        -92,
        37
    ]
}
POINT (-92 37)
```

Notice that the `geojson` module has an interface for different data types and saves us from setting the `type` and `coordinates` attributes manually. Now, imagine if you had a geographic object with hundreds of features. You could programmatically build this data structure instead of building a very large string.

The `geojson` module is also the reference implementation for the Python `geo_interface` convention. This interface allows cooperating programs to exchange data seamlessly and in a Pythonic way without the programmer explicitly exporting and importing GeoJSON strings. So, if we wanted to feed the point we created with the `geojson` module to the `Shapely` module, we could perform the following command, which reads the `geojson` module's point object straight into Shapely, after which we'll export it as WKT:

```
from shapely.geometry import asShape  
point = asShape(p)  
Point.wkt
```

This will output the following WKT string:

```
'POINT (-92.00000000000000 37.00000000000000)'
```

More and more geospatial Python libraries are implementing both the `geojson` and `geo_interface` functionalities, including PyShp, Fiona, Karta, and ArcGIS. Third-party implementations exist for QGIS.

GeoJSON is a simple text format that is human- and computer-readable. Now, we'll look at some binary vector formats.

## OGR

We touched on OGR as a way to handle WKT strings, but its real power is as a universal vector library. This book strives for pure Python solutions, but no single library even comes close to the variety of formats that OGR can process.

OGR is contained in the GDAL library, which we can install with conda:

```
conda install gdal
```

Let's read an example point shapefile using the OGR Python API. The example shapefile can be downloaded as a ZIP file from <https://github.com/GeospatialPython/Learn/raw/master/point.zip>.

This point shapefile has five points with single-digit, positive coordinates. The attributes list the order in which the points were created, making it useful for testing. This simple example will read in the point shapefile and loop through each feature; then, it will print the x and y values of each point, plus the value of the first attribute field:

```
from osgeo import ogr  
shp = ogr.Open("point.shp")  
layer = shp.GetLayer()  
for feature in layer:
```

```
... geometry = feature.GetGeometryRef()
... print(geometry.GetX(), geometry.GetY(), feature.GetField("FIRST_"
FLD"))
...
...
```

The output is as follows:

```
1.0 1.0 First
3.0 1.0 Second
4.0 3.0 Third
2.0 2.0 Fourth
0.0 0.0 Appended
```

This example is simple, but OGR can become quite verbose as your script becomes more complex. Next, we'll look at a simpler way to deal with shapefiles.

## PyShp

PyShp is a simple, pure Python library that reads and writes shapefiles. It doesn't perform any geometry operations and only uses Python's standard library. It's contained in a single file that's easy to move around, squeeze onto small, embedded platforms, and modify. It is also compatible with Python 3. It also implements `geo_interface`. The PyShp module is available through conda as `pyshp`.

Let's repeat the previous OGR example with PyShp:

```
import shapefile
shp = shapefile.Reader("point.shp")
for feature in shp.shapeRecords():
... point = feature.shape.points[0]
... rec = feature.record[0]
... print(point[0], point[1], rec)
...
...
```

We'll see the following attributes:

```
1.0 1.0 First
3.0 1.0 Second
4.0 3.0 Third
2.0 2.0 Fourth
0.0 0.0 Appended
```

## Shapely

Shapely was mentioned in the *Well-Known Text (WKT)* section for its import and export functions. However, its true purpose is as a generic geometry library. Shapely is a high-level, Pythonic interface to the GEOS library for geometric operations. In fact, Shapely intentionally avoids reading or writing files. It relies completely on data import and export from other modules and maintains its focus on geometry manipulation. You can install `shapely` using conda.

Let's do a quick Shapely demonstration in which we'll define a single WKT polygon and then import it into Shapely. Then, we'll measure the area. Our computational geometry will consist of buffering that polygon by a measure of five arbitrary units, which will return a new, bigger polygon for which we'll measure the area:

```
from shapely import wkt
wktPoly = "POLYGON((0 0,4 0,4 4,0 4,0 0))"
poly = wkt.loads(wktPoly)
poly.area 16.0
buf = poly.buffer(5.0)
buf.area 174.41371226364848
```

We can then perform a difference in the area of the buffer and the original polygon area, as shown here:

```
buf.difference(poly).area
158.413712264
```

If you can't have pure Python, a Pythonic API as clean as Shapely that packs such a punch is certainly the next best thing.

## Fiona

The Fiona library provides a simple Python API around the OGR library for data access and nothing more. This approach makes it easy to use and is less verbose than OGR while using Python. Fiona outputs GeoJSON by default. You can install `fiona` through conda.

As an example, we'll use the `GIS_CensusTract_poly.shp` file from the `dbfpy` example we looked at earlier in this chapter.

First, we'll import `fiona` and Python's `pprint` module to format the output. Then, we'll open the shapefile and check its driver type:

```
import fiona
from pprint import pprint
f = fiona.open("GIS_CensusTract_poly.shp")
f.driver
ESRI Shapefile
```

Next, we'll check its coordinate reference system and get the data bounding box, as shown here:

```
f.crs  
{'init': 'epsg:4269'}  
f.bounds
```

We'll see the coordinates of the bounding box as follows:

```
(-89.8744162216216, 30.161122135135138, -89.1383837783784,  
30.661213864864862)
```

Now, we'll view the data schema as `geojson` and format it using the `pprint` module, as you can see in the following lines of code:

```
pprint(f.schema)  
{'geometry': 'Polygon',  
'properties': {'GEODB_OID': 'float:11', 'OBJECTID': 'float:11',  
'PERMANEO': 'str:40',  
'SOURCE_1': 'str:40',  
'SOURCE_2': 'str:40',  
'SOURCE_3': 'str:100',  
'SOURCE_4': 'str:130',  
'DATA_SE5': 'str:46',  
'DISTRIB6': 'str:188',  
'LOADDATE': 'date',  
'QUALITY': 'str:35',  
'SCALE': 'str:52',  
'FCODE': 'str:38',  
'STCO_FI7': 'str:5',  
'STATE_NAME': 'str:140',  
'COUNTY_8': 'str:60',  
'CENSUST9': 'str:20',  
'POPULAT10': 'float:11',  
'AREASQKM': 'float:31.15',  
'GNIS_ID': 'str:10',  
'POPULAT11': 'float:11',  
'DB2GSE_12': 'float:31.15',  
'DB2GSE_13': 'float:31.15'}}}
```

Next, let's get a count of the number of features:

```
len(f)  
45
```

Finally, we'll print one of the records as formatted GeoJSON, as shown here:

```
pprint(f[1])
{'geometry': {'coordinates': [[[(-89.86412366375093,
30.661213864864862), (-89.86418691770497, 30.660764012731285),
(-89.86443391770518, 30.659652012730202),
...
'type': 'MultiPolygon'},
'id': '1',
'properties': {'GEODB_OID': 4360.0,
'OBJECTID': 4360.0,
'PERMANEO': '9a914eef-9249-44cf-a05f-af4b48876c59', 'SOURCE_1': 'NA',
'SOURCE_2': '20006',
...
'DB2GSE_12': 351242560.967882,
'DB2GSE_13': 101775.283967268},
'type': 'Feature'}
```

## GDAL

GDAL is the dominant geospatial library for raster data. Its raster capability is so significant that it is a part of virtually every geospatial toolkit in any language, and Python is no exception to this. To see the basics of how GDAL works in Python, download the following example raster satellite image as a ZIP file and unzip it: <https://github.com/GeospatialPython/Learn/raw/master/SatImage.zip>. Let's open this image and see how many bands it has and how many pixels are present along each axis:

```
from osgeo import gdal
raster = gdal.Open("SatImage.tif")
raster.RasterCount 3
raster.RasterXSize 2592
raster.RasterYSize 2693
```

GDAL is an extremely fast geospatial raster reader and writer within Python. It can also reproject images quite well in addition to being able to do a few other tricks. However, the true value of GDAL comes from its interaction with the next Python module, which we'll examine now.

## NumPy

NumPy is an extremely fast, multidimensional Python array processor designed specifically for Python and scientific computing but is written in C. It is available via PyPI or as a wheel file (available at <http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy>) and can be installed with ease. In addition to its amazing speed, the magic of NumPy includes its interaction with other libraries.

NumPy can exchange data with GDAL, Shapely, the **Python Imaging Library (PIL)**, and many other scientific computing Python libraries in other fields.

As a quick example of NumPy's ability, we'll combine it with GDAL to read in our example satellite image and then create a histogram of it. The interface between GDAL and NumPy is a GDAL module called `gdal_array`, which has NumPy as a dependency. Numeric is the legacy name of the NumPy module. The `gdal_array` module imports `numpy`.

In the following example, we'll use `gdal_array`, which imports `numpy`, to read the image in as an array, grab the first band, and save it as a JPEG image:

```
from osgeo import gdal_array
srcArray = gdal_array.LoadFile("SatImage.tif")
band1 = srcArray[0]
gdal_array.SaveArray(band1, "band1.jpg", format="JPEG")
```

This code produces the following image, which appears as black and white because we only have the blue band of the image without the red and green bands to give it full RGB color:



Figure 4.2 – A single band of a satellite image extracted with `gdal_array`

Now let's look at other ways to work with raster data starting with PIL.

## PIL

PIL was originally developed for remote sensing but has evolved into a general image-editing library for Python. Like NumPy, it is written in C for speed but is designed specifically for Python. In addition to image creation and processing, it also has a useful raster drawing module. PIL is also available via PyPI; however, in Python 3, you may want to use the Pillow module, which is an upgraded version of PIL. As you'll see in the following example, we can use a Python `try` statement to import PIL using two possible variations, depending on how you installed it.

In this example, we'll combine PyShp and PIL to rasterize the hancock shapefile from the previous examples and save it as an image. We'll use a world-to-pixel coordinate transformation similar to our SimpleGIS from *Chapter 1, Learning about Geospatial Analysis with Python*. We'll create an image to use as a canvas in PIL, and then we'll use the PIL ImageDraw module to render the polygon. Finally, we'll save it as a PNG image, as you can see in the following lines of code:

```
try:  
    import Image  
    import ImageDraw  
except:  
    from PIL import Image  
    from PIL import ImageDraw  
import shapefile  
r = shapefile.Reader("hancock.shp")  
xdist = r.bbox[2] - r.bbox[0]  
ydist = r.bbox[3] - r.bbox[1]  
iwidth = 400  
iheight = 600  
xratio = iwidth/xdist  
yratio = iheight/ydist  
pixels = []  
for x,y in r.shapes()[0].points:  
    px = int(iwidth - ((r.bbox[2] - x) * xratio))  
    py = int((r.bbox[3] - y) * yratio)  
    pixels.append((px,py))  
img = Image.new("RGB", (iwidth, iheight), "white")  
draw = ImageDraw.Draw(img)  
draw.polygon(pixels, outline="rgb(203, 196, 190)", 204, 189)")  
img.save("hancock.png")
```

This example creates the following image:

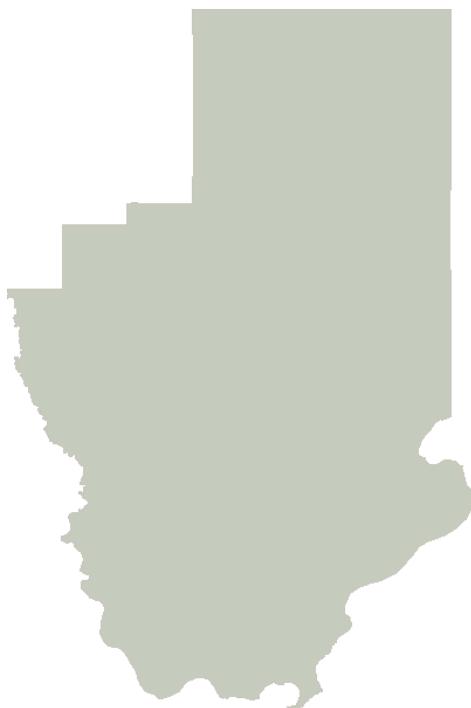


Figure 4.3 – A rasterized version of a vector shapefile using PIL

Now we'll look at a pure Python way of creating simple images.

## PNGCanvas

Sometimes, you may find that PIL is overkill for your purposes, or you are not allowed to install PIL because you do not have administrative rights to the machine that you're using to install Python modules that have been created and compiled in C. In those cases, you can usually get away with the lightweight pure Python `PNGCanvas` module.

`PNGCanvas` is one of the specialized libraries not included in Anaconda so we'll have to install it using `pip`:

```
pip install pngcanvas
```

Using this module, we can repeat the raster shapefile example we performed using PIL but in pure Python, as you can see here:

```
import shapefile  
import pngcanvas
```

```
r = shapefile.Reader("hancock.shp")
xdist = r.bbox[2] - r.bbox[0]
ydist = r.bbox[3] - r.bbox[1]
iwidth = 400
iheight = 600
xratio = iwidth/xdist
yratio = iheight/ydist
pixels = []
for x,y in r.shapes()[0].points:
    px = int(iwidth - ((r.bbox[2] - x) * xratio))
    py = int((r.bbox[3] - y) * yratio)
    pixels.append([px,py])
c = pngcanvas.PNGCanvas(iwidth,iheight)
c.polyline(pixels)
f = open("hancock_pngcvs.png", "wb")
f.write(c.dump())
f.close()
```

This example gives us a simple outline as PNGCanvas does not have a built-in `fill` method:

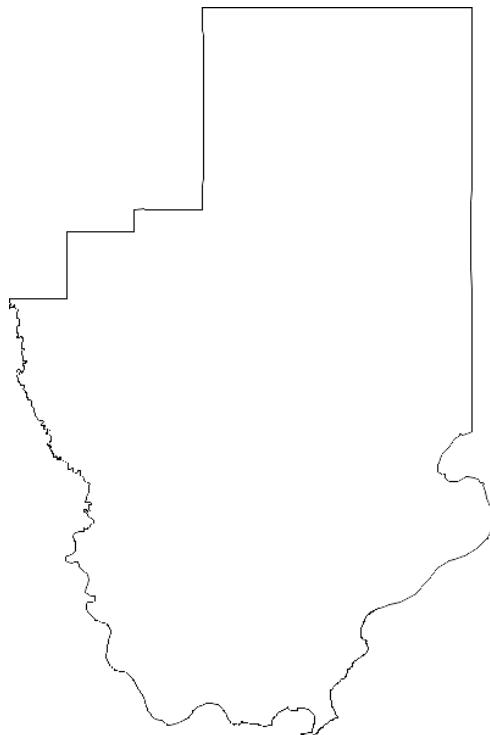


Figure 4.4 – A rasterized shapefile using the pure Python PNGCanvas library

Next, let's look at a very powerful way of rendering geospatial data using one of the larger libraries we'll see in this book.

## GeoPandas

`pandas` is a high-performance Python data analysis library that can handle large datasets that are tabular (similar to a database), ordered/unordered, labeled matrices, or unlabeled statistical data. `GeoPandas` is simply a geospatial extension to `pandas` that builds upon `Shapely`, `Fiona`, `PyProj`, `matplotlib`, and `Descartes`, all of which must be installed. It allows you to easily perform operations in Python that would otherwise require a spatial database such as PostGIS.

As `GeoPandas` has so many complex dependencies, we'll create a separate environment for it in Anaconda to keep things clean. First we need to deactivate our current environment:

```
conda deactivate lgawp4
```

Next we'll create a new one for `GeoPandas`:

```
conda create -n geopandas python=3.10.9
```

Then we can activate it as follows:

```
conda activate geopandas
```

Then we need to install `GeoPandas` itself:

```
conda install geopandas
```

The following script opens a shapefile and dumps it into GeoJSON. Then, it creates a map with `matplotlib`:

```
import geopandas
import matplotlib.pyplot as plt
gdf = geopandas.GeoDataFrame
census = gdf.from_file("GIS_CensusTract_poly.shp")
census.plot()
plt.show()
```

The following is the resulting map plot from the previous commands:

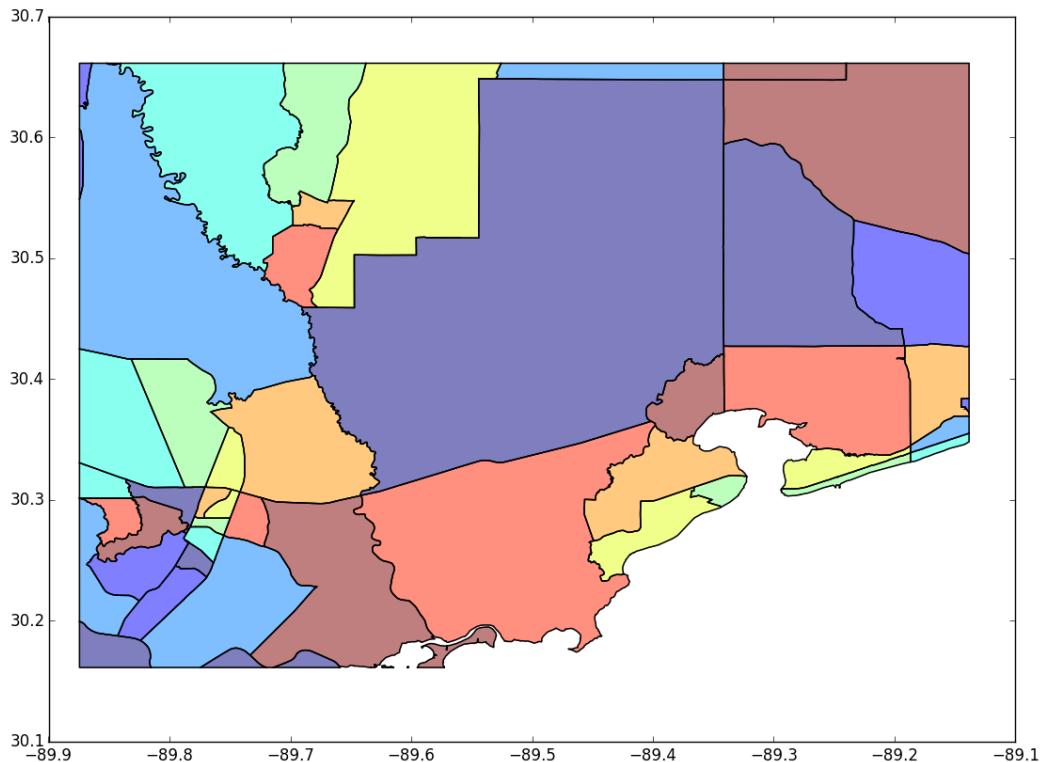


Figure 4.5 – Shapefile plotted in matplotlib using GeoPandas

Now let's look at one more way to render data – as a PDF map.

## PyFPDF

The pure Python PyFPDF library is a lightweight way to create PDFs, including maps. Because the PDF format is a widely used standard, PDFs are commonly used to distribute maps. You can install it via PyPI as `fpdf`.

The official name of the software is PyFPDF because it is a part of the PHP language module called `fpdf`. This module uses a concept called a cell to lay items out at specific locations on a page. As a quick example, we'll import the `hancock.png` image we created in the `PIL` example into a PDF called `map.pdf` to create a simple PDF map. The map will have header text at the top that says Hancock County Boundary, followed by the map image:

```
import fpdf  
# PDF constructor:
```

```
# Portrait, millimeter units, A4 page size
pdf=fpdf.FPDF("P", "mm", "A4")
# create a new page
pdf.add_page()
# Set font: arial, bold, size 20
pdf.set_font('Arial','B',20)
# Layout cell: 160 x 25mm, title, no border, centered
pdf.cell(160,25,'Hancock County Boundary', \
border=0, align="C")
# Write the image specifying the size
pdf.image("hancock.png",25,50,110,160)
# Save the file: filename, F = to file System
pdf.output('map.pdf', 'F')
```

If you open the PDF file named `map.pdf` in Adobe Acrobat Reader or another PDF reader such as Sumatra PDF, you'll see that the image is now centered on an A4 page. Geospatial products are often included as part of larger reports, and the PyFPDF module simplifies automatically generating reports as PDFs.

Now that we can render data as images, let's look at a powerful way to query data using a geospatial database.

## PyMySQL

The popular MySQL (available at <http://dev.mysql.com/downloads>) database system is gradually evolving spatial functions. It has support for OGC geometries and a few spatial functions. It also has a pure Python API available in the PyMySQL library. The limited spatial functions use planar geometry and bounding rectangles as opposed to spherical geometry and shapes. The latest development release of MySQL contains some additional functions that improve this capability.

In the following example, we'll create a database in MySQL called `spatial_db`. Then, we'll add a table called `PLACES` with a geometry column. Next, we'll add two cities as point locations. Finally, we'll calculate the distance using MySQL's `ST_Distance` function and then convert the result from degrees into miles. In order to perform this example, you'll have to install MySQL following the instructions on the download site.

First, we will import our `mysql` library and set up the database connection:

```
# Import the python mysql library import pymysql
# Establish a database connection on our local # machine as the root
database user.
conn = pymysql.connect(host='localhost', port=3306, user='root',
passwd='', db='mysql')
```

Next, we get the database cursor:

```
# Get the database cursor needed to change # the database
cur = conn.cursor()
```

Now, we check whether the database already exists, and drop it if it does:

```
# If the database already exists, delete
# it and recreate it either way.
cur.execute("DROP DATABASE IF EXISTS spatial_db") cur.execute("CREATE
DATABASE spatial_db")
# Close the cursor and the connection cur.close()
conn.close()
```

Now, we set up a new connection and get a cursor:

```
# Set up a new connection and cursor
conn = pymysql.connect(host='localhost', port=3306, user='root',
passwd='', db='spatial_db')
cur = conn.cursor()
```

Next, we can create our new table and add our fields:

```
# Create our geospatial table
# Add name and location fields. The location
# field is spatially enabled to hold GIS data
cur.execute("CREATE TABLE PLACES (id int NOT NULL
AUTO_INCREMENT PRIMARY KEY, Name varchar(50) NOT NULL, location
Geometry NOT NULL)")
```

With the fields added, we are ready to insert records for the location of some cities:

```
# Insert a name and location for the city of
# New Orleans
cur.execute("INSERT INTO PLACES (name, location) VALUES ('NEW
ORLEANS', GeomFromText('POINT(30.03 90.03)'))")
# Insert a name and location for the city of
# Memphis.
cur.execute("INSERT INTO PLACES (name, location) VALUES ('MEMPHIS',
GeomFromText('POINT(35.05 90.00)'))")
```

Then, we can commit changes to the database:

```
# Commit the changes to the database
conn.commit()
```

Now, we can query the database! First, we'll get a list of all of the point locations:

```
# Now let's read the data. Select all of # the point locations from
# the database.
cur.execute("SELECT AsText(location) FROM PLACES")
```

Now, we'll extract the two points from the query results:

```
# We know there's only two points, so we'll
# just parse them.
p1, p2 = [p[0] for p in cur.fetchall()]
```

Before we can measure the distance, we need to convert the point listings into geospatial geometries:

```
# Now we'll convert the data
# to geometries to measure the distance
# between the two cities
cur.execute("SET @p1 = ST_GeomFromText('{}')".format(p1)) cur.
execute("SET @p2 = ST_GeomFromText('{}')".format(p2))
```

Finally, we can use the *Distance* stored procedure to measure the distance between the two geometries:

```
# Now we do the measurement function which # is also a database query.
cur.execute("SELECT ST_Distance(@p1, @p2)") d = float(cur.fetchone()
[0])
# Print the distance as a formatted # string object.
print("{:.2f} miles from New Orleans to Memphis".format(d * 70))
cur.close()
conn.close()
```

The output is as follows:

```
351.41 miles from New Orleans to Memphis
```

### Using other spatial engines

There are other spatial database options available, including PostGIS and SpatiaLite; however, Python 3 support for these spatial engines is developmental at best. You can access PostGIS and MySQL through the OGR library; however, MySQL support is limited.

MySQL lets us work with vector data in a database. Next let's experiment with a library that lets us read geospatial raster data.

## Rasterio

The GDAL library we introduced earlier in this chapter is extremely powerful, but it wasn't designed for Python. The `rasterio` library solves that problem by wrapping GDAL in a very simple, clean Pythonic API for raster data operations.

This example uses the satellite image from the GDAL example in this chapter. We'll open the image and get some metadata as follows:

```
import rasterio
ds = rasterio.open("SatImage.tif")
ds.name 'SatImage.tif'
ds.count 3
ds.width 2592
ds.height 2693
```

Next up, we'll examine a powerful library that works with the crowd-sourced Open Street Map data and can even do routing.

## OSMnx

The `osmnx` library combines **Open Street Map (OSM)** and the powerful `NetworkX` library to manage street networks used for routing. This library has dozens of dependencies that it rolls up to handle all of the complex steps of downloading, analyzing, and visualizing street networks.

You can install `osmnx` using pip:

```
pip install osmnx
```

The following example uses `osmnx` to download street data from OSM for a city, creates a street network from it, and calculates some basic statistics:

```
import osmnx as ox
G = ox.graph_from_place('Bay Saint Louis, MS , USA', network_
type='drive')
stats = ox.basic_stats(G)
stats["street_length_avg"] 172.1468804611654
```

Since we're talking about web-based data sources, let's look at a great way to create web-based maps next!

## Folium

Folium is a Python library that makes it easy to generate web maps using the `Leaflet` web-mapping library through Python code without editing any HTML or JavaScript normally required when working with `Leaflet`. Once the web map is created, you can still customize it further by editing the resulting HTML. Folium supports a number of overlay data types so you can add data to the included basemap and build interactive, thematic GIS maps.

You can install Folium through Conda using the following:

```
conda install folium
```

This short script creates a map centered on a given location at zoom level 13 out of 16, then adds an pop-up informational marker with some text:

```
import folium
m = folium.Map(location=[30.3088, -89.3300], zoom_start=13)
folium.Marker(
    location=[30.32, -89.3300],
    popup="A Place Apart",
    icon=folium.Icon(color="green"),
).add_to(m)
m.save("map.html")
```

The script saves the web map to the `map.html` file, which you can open in a web browser by double-clicking it on your filesystem. The map output looks like the following:

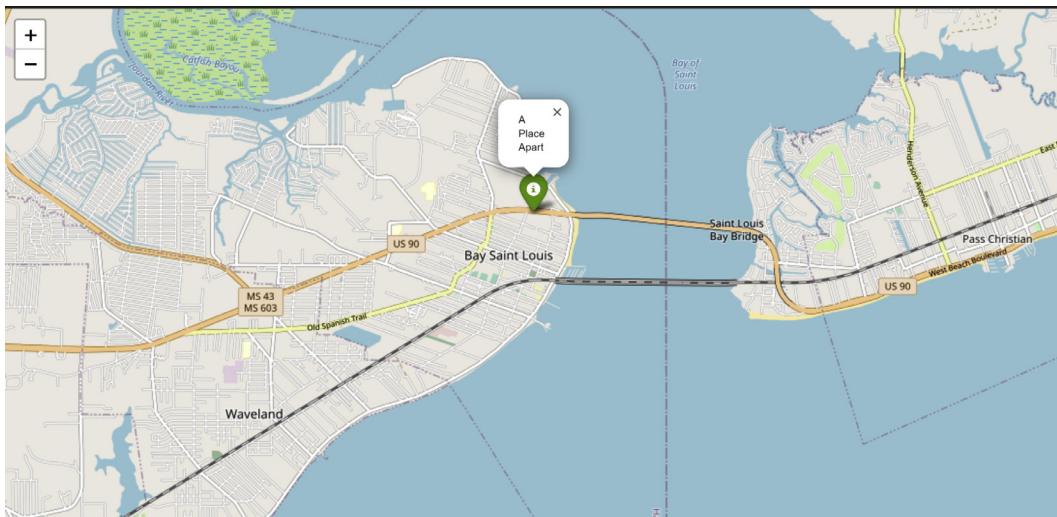


Figure 4.6 – A simple web map with an interactive informational marker created with Folium in Python

In addition to markers and polylines, Folium can also overlay `geojson` and `topojson` data.

## Summary

In this chapter, we surveyed the Python-specific tools for geospatial analysis. We looked at an easier way to install third-party modules using Anaconda. We reviewed the libraries needed for networking including `urllib`, `requests`, and `ftplib`. We experimented with bundling, compressing, and decompressing geospatial files with the `zipfile` and `tarfile` modules. We parsed and created tag-based data with `minidom` and `ElementTree`. We manipulated WKT data strings with `shapely`. We worked with JSON-based data using `JSON` and `GeoJSON`.

---

Later, we also learned about the most powerful vector library, OGR, and the pure-Python shapefile library `PyShp`. We learned how to access geospatial data using the Pythonic `fiona` library. We examined the powerful GDAL raster data library and its connection to Python's large array math library NumPy. We learned how to create map images with PIL and PNGCanvas as well as GeoPandas. We created PDF maps using PyFPDF. We made a geospatial database using PyMySQL. We accessed raster data with the Pythonic `rasterio` module. We learned how to do traffic routing with OSMnx and how to quickly make web maps with Folium.

Many of these tools included bindings to the libraries we discussed in the previous chapter as best-of-breed solutions for specific operations, such as GDAL's raster access functions. We also included pure Python libraries as much as possible and will continue to include pure Python algorithms as we work through the upcoming chapters.

In the next chapter, we'll begin applying all of these tools for GIS analysis.

## Questions

1. What are the two major type systems to manage Python libraries?
2. Which is the powerful Python library that can perform math on large arrays of numbers, which is useful for remote sensing algorithm math?
3. What are the three relational database systems that have spatial engines available to Python?

## Further reading

- <https://www.anaconda.com/>
- <https://jupyter.org/>
- <https://gdal.org/>
- <https://numpy.org/>
- <https://diveintopython3.net/>



# 5

## Python and Geospatial Algorithms

This chapter will focus on applying Python to algorithms that are typically performed by a **geographic information system (GIS)** such as QGIS or ArcGIS. An algorithm is like a recipe for a computer. Just like a recipe gives you step-by-step instructions on how to cook a dish, an algorithm gives a computer step-by-step instructions to solve a problem or complete a task. These algorithms are the heart and soul of geospatial analysis. We will continue to use as few external dependencies as possible outside of Python itself so that you have tools with maximum reusability in different environments. In this book, we separate GIS analysis and remote sensing from a programming perspective, which means that, in this chapter, we'll mostly focus on vector data.

As with the other chapters in this book, the items presented here are core functions that serve as building blocks that you can recombine to solve challenges that you will encounter beyond this book. The topics in this chapter include the following:

- Measuring distance
- Calculating line direction
- Understanding coordinate conversion
- Understanding reprojection
- Understanding coordinate format conversion
- Calculating the area of a polygon
- Using ChatGPT to measure a polygon's perimeter

This chapter contains many code samples. It covers important algorithms mostly related to measuring distance. Relating objects on Earth by how close they are physically is a core tenant of GIS.

## Technical requirements

You can find the code files for this chapter on GitHub at [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730\\_05\\_Asset\\_Files](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730_05_Asset_Files).

## Measuring distance

The essence of geospatial analysis is discovering the relationships between objects on Earth. Items that are closer together tend to have a stronger relationship than those that are farther apart. This concept is known as Tobler's First Law of Geography. Therefore, measuring distance is a critical function of geospatial analysis.

Every map is a model of the Earth and they are all wrong to some degree due to map projection compromises, data resolution, scale, timelines, or other factors. For this reason, measuring the accurate distance between two points on the Earth while sitting in front of a computer is impossible. Even professional land surveyors (who go out in the field with both traditional sighting equipment and very precise GPS equipment) fail to account for every anomaly in the Earth's surface between point A and point B. So, to measure distance, we must look at the following questions:

- What are we measuring?
- How much are we measuring?
- How much accuracy do we need?

Now, to calculate distance, there are three models of the Earth that we can use:

- Flat plane
- Spherical
- Ellipsoid

In the flat plane model, standard Euclidean geometry is used. The Earth is considered a flat plane with no curvature, as shown in the following diagram:

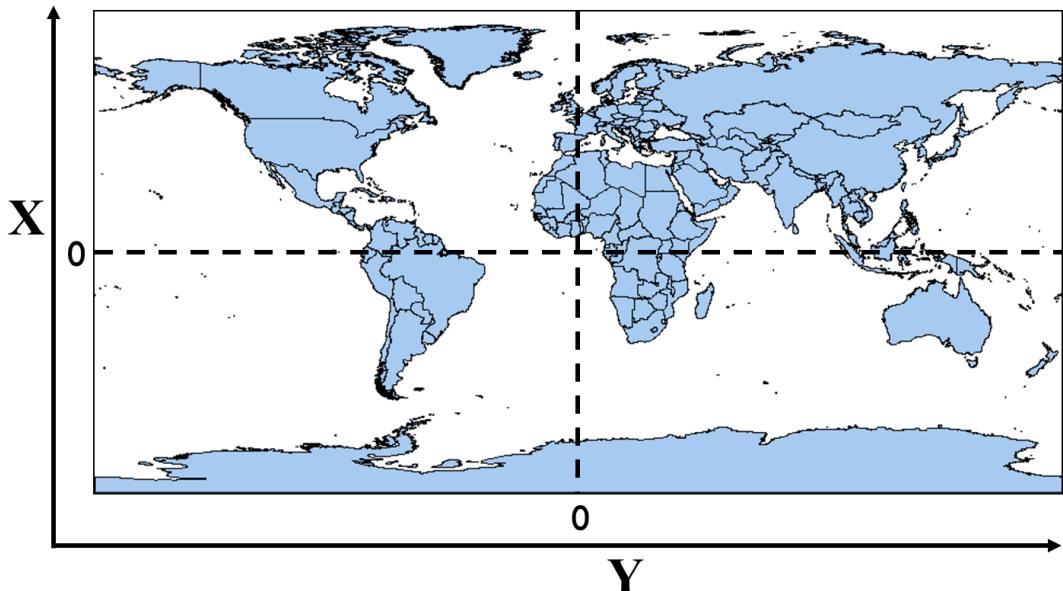


Figure 5.1 – The flat plane model treats the Earth as a flat plane with the equator as the Y origin and the prime meridian as the X origin

This model makes math quite simple because you work with straight lines. The most common format for geospatial coordinates is decimal degrees. However, decimal degree coordinates are reference measurements on a sphere taken as angles – between the longitude and the prime meridian—and the latitude and equator. Furthermore, the lines of longitude converge toward zero at the poles. The circumference of each line of latitude becomes smaller toward the poles as well. These facts mean decimal degrees are not a valid coordinate system for Euclidean geometry, which uses infinite planes.

Map projections attempt to simplify the issues of dealing with a 3D ellipsoid in a 2D plane, either on paper or on a computer screen. As we discussed in *Chapter 1, Learning about Geospatial Analysis with Python*, map projections flatten a round model of the Earth to a plane and introduce distortion in exchange for the convenience of a map. Once this projection is in place and decimal degrees are traded for a Cartesian coordinate system with  $x$  and  $y$  coordinates, we can use the simplest forms of Euclidean geometry—namely, the Pythagorean theorem.

At a large enough scale, a sphere or ellipsoid such as the Earth appears more like a plane than a sphere. In fact, for centuries, everyone thought the Earth was flat! If the difference in degrees of longitude is small enough, you can often get away with using Euclidean geometry and then converting the measurements into meters, kilometers, or miles. This method is generally not recommended but the decision is ultimately up to you and your requirements for accuracy as an analyst.

The spherical model approach tries to better approximate reality by avoiding the problems resulting from smashing the Earth onto a flat surface. As the name suggests, this model uses a perfect sphere for representing the Earth (similar to a physical globe), which allows us to work with degrees directly. This model ignores the fact that the Earth is really more of an egg-shaped ellipsoid with varying degrees of thickness in its crust. But by working with distance on the surface of a sphere, we can begin to measure longer distances with more accuracy. The following screenshot illustrates this concept:

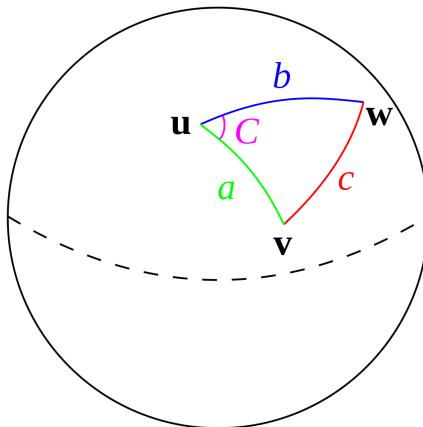


Figure 5.2 – The spherical model allows us to more accurately measure longer distances and work in degrees directly

Using the ellipsoid model of the Earth, analysts strive for the best model of the Earth's surface. There are several ellipsoid models, which are called datums. A **datum** is a set of values that define an estimated shape for the Earth, also known as a **geodetic system**. Like any other georeferencing system, a datum can be optimized for a localized area. The most commonly used datum is called **WGS84**, which is designed for global use. You should be aware that WGS84 is occasionally updated as assessment techniques and technology improves. The most recent revision occurred in 2004.

In North America, the NAD83 datum is used to optimize referencing over the continent. In the eastern hemisphere, the **European Terrestrial Reference System 1989 (ETRS89)** is used more frequently. ETRS89 is fixed to the stable part of the **Eurasian Plate**. Maps of Europe based on ETRS89 are immune to continental drift, which changes up to 2.5 cm per year as the Earth's crust shifts.

An ellipsoid does not have a constant radius from the center. This fact means the formulas used in the spherical model of the Earth begin to have issues in the ellipsoid model. Though not a perfect approximation, it is much closer to reality than the spherical model.

The following screenshot shows a generic ellipsoid model denoted by a black line contrasted against a representation of the Earth's uneven crust, which is using a red line to represent the geoid. Although we will not use it for these examples, another model is the geoid model. The geoid is the most precise and accurate model of the Earth, which is based on the Earth's surface with no influencing factors except

gravity and rotation. The following diagram is a representation of a geoid, ellipsoid, and spherical model to illustrate their differences:

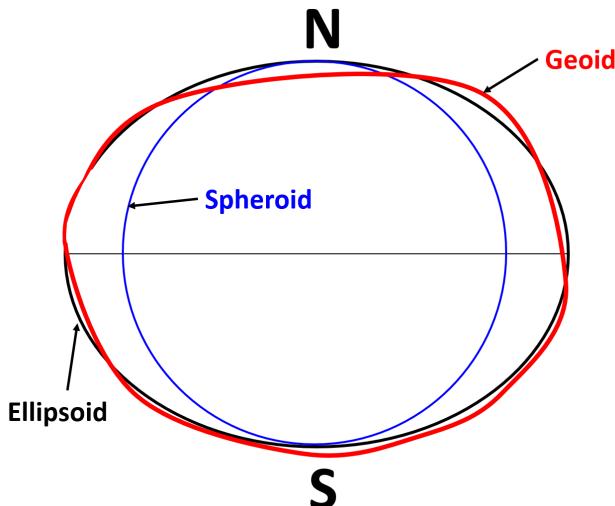


Figure 5.3 – Differences in shape between the spheroid, ellipsoid, and geoid Earth models

Understanding these models of the Earth is critical to everything else in this book because we're modeling the Earth, after all.

Now that we've discussed the different models of the Earth and the issues with measuring them, let's look at some solutions using Python.

## Using the Pythagorean theorem to measure distance

We'll start by measuring with the simplest method, that is, the Pythagorean theorem, also known as Euclidean distance. If you remember your geometry lessons from school, the Pythagorean theorem asserts the following:

$$a^2 + b^2 = c^2$$

In this assertion, the  $a$ ,  $b$ , and  $c$  variables are all sides of a right-angle triangle (with  $c$  being the hypotenuse). You can solve for any one side if you know the other two.

In this example, we'll start with two projected points in **the Mississippi Transverse Mercator (MSTM)** projection. The units of this projection are in meters. The  $x$  axis locations are measured from the central meridian defined by the westernmost location in the state.

The  $y$  axis is defined from the NAD83 horizontal datum. The first point, defined as  $(x_1, y_1)$ , represents Jackson, the state capital of Mississippi. The second point, defined as  $(x_2, y_2)$ , represents the city of Biloxi, which is a coastal town, as shown in the following illustration:

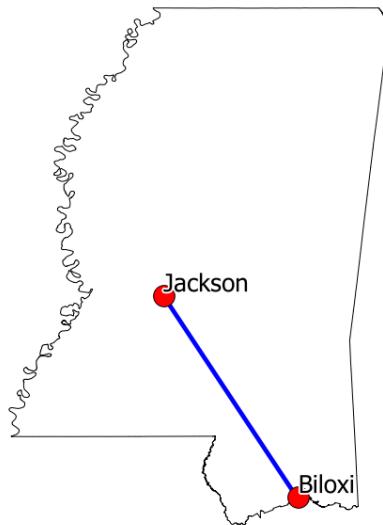


Figure 5.4 – The Pythagorean theorem can be used to measure distance between two points on a map

We'll import the Python `math` module for its square root function called `sqrt()`. Then, we'll calculate the *x* axis and *y* axis distances. Finally, we'll use these variables to execute the Euclidean distance formula to get the distance across the bounding box in meters from an *x*, *y* origin, which will be used in the MSTM projection:

```
import math
x1 = 456456.23
y1 = 1279721.06
x2 = 576628.34
y2 = 1071740.33
x_dist = x1 - x2
y_dist = y1 - y2
# Pythagorean Theorem
dist_sq = x_dist**2 + y_dist**2
distance = math.sqrt(dist_sq)
print(distance)
```

The output should be `240202.66`. So, the distance is approximately 240,202 meters, which is around 240.2 kilometers or 150 miles. This calculation is reasonably accurate because this projection is optimized for measuring distance and area in Mississippi using Cartesian coordinates.

We can also measure distance using decimal degrees, but we must perform a few additional steps. To measure using degrees, we must convert the angles into radians, which account for the curved surface distance between the coordinates. We'll also multiply our output in radians times the radius of the Earth in meters to convert back from radians.

### More about radians

You can read more about radians at <http://en.wikipedia.org/wiki/Radian>.

We'll perform this conversion using the Python `math.radians()` method in the following code when we calculate the *x* and *y* distances:

```
x1 = -90.21
y1 = 32.31
x2 = -88.95
y2 = 30.43
x_dist = math.radians(x1 - x2)
y_dist = math.radians(y1 - y2)
dist_sq = x_dist**2 + y_dist**2
dist_rad = math.sqrt(dist_sq)
```

We'll multiply the radians by 6,371 kilometers, which is the Earth's radius:

```
print(dist_rad * 6371)
```

Okay, so this time, we came up with around 251 kilometers, which is 11 kilometers more than our first measurement. So, as you can see, your choice of measurement algorithm and Earth model can have significant consequences. Using the same equation, we come up with radically different answers, depending on our choice of coordinate system and Earth model.

### More about Euclidean distance

You can read more about Euclidean distance at <http://mathworld.wolfram.com/Distance.html>.

Let's check out the haversine formula next.

## Using the haversine formula

Part of the problem with using the Pythagorean theorem to measure distance on the Earth, which is a sphere, is the concept of great circle distance. A great circle is the shortest distance between two points on a sphere. Another important feature that defines a great circle is that the circle, if followed all of the way around the sphere, will bisect the sphere into two equal halves, as shown in the following Wikipedia illustration (taken from [https://en.wikipedia.org/wiki/Great\\_circle#/media/File:Great\\_circle\\_hemispheres.png](https://en.wikipedia.org/wiki/Great_circle#/media/File:Great_circle_hemispheres.png)):

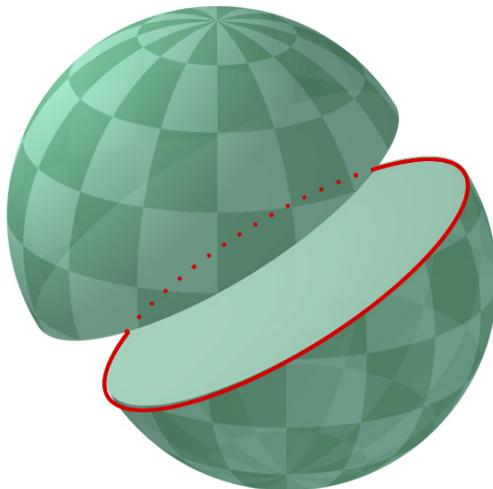


Figure 5.5 – A great circle dissects the sphere into symmetrical halves wherever it is drawn

So, what is the right way to measure a line on a curved sphere? The most popular method is to use the haversine formula, which uses trigonometry to calculate the Great Circle distance using coordinates defined in decimal degrees as input. The haversine formula is  $\text{haversine}(\theta) = \sin^2(\theta/2)$ , where  $\theta$  is the central angle between two points on a sphere. Once again, we'll convert the axis distances from degrees into radians before we apply the formula, just like in the previous example. But this time, we'll also convert the latitude ( $y$  axis) coordinates into radians separately:

```
import math
x1 = -90.21
y1 = 32.31
x2 = -88.95
y2 = 30.43
x_dist = math.radians(x1 - x2)
y_dist = math.radians(y1 - y2)
y1_rad = math.radians(y1)
y2_rad = math.radians(y2)
a = math.sin(y_dist/2)**2 + math.sin(x_dist/2)**2 \
    * math.cos(y1_rad) * math.cos(y2_rad)
c = 2 * math.asin(math.sqrt(a))
distance = c * 6371 # kilometers
print(distance)
```

The output is 240.84. We get 240.6 kilometers using the haversine formula, compared to 240.2 kilometers using the optimized and more accurate projection. This difference is less than half a kilometer, which is not bad for a distance calculation of two cities 150 miles apart. The haversine formula is the most commonly used distance measuring formula because it is relatively lightweight

from a coding perspective and reasonably accurate in most cases. It is considered to be accurate to within about a meter.

To summarize what we've learned so far, most of the point coordinates you encounter as an analyst are in unprojected decimal degrees. So, your options for measurement are as follows:

- Reproject to a distance-accurate Cartesian projection and measure
- Just use the haversine formula and see how far it takes you for your analysis
- Use the even more precise Vincenty formula

That's right! There's another formula that seeks to provide an even better measurement than haversine. Let's see how accurate we can get using the Vincenty formula.

## Using the Vincenty formula

So, we've examined distance measurement using the Pythagorean theorem (flat Earth model) and the Haversine formula (spherical Earth model). The Vincenty formula accounts for the ellipsoid model of the Earth, which is more accurate. And if you are using a localized ellipsoid, it can be accurate to much less than a meter.

In the following implementation of this formula, you can change the semi-major axis value and flattening ratio to fit the definition of any ellipsoid. Let's see what the distance is when we measure using the Vincenty formula on the NAD83 ellipsoid in the following example. This example requires more code than the previous examples so we'll step through it:

1. Import the `math` library to access essential mathematical functions required for trigonometric and square root calculations:

```
import math
```

2. Define the coordinates (latitude and longitude) of the two points for which you want to measure the distance, and initialize the `distance` variable:

```
distance = None
x1 = -90.21
y1 = 32.31
x2 = -88.95
y2 = 30.43
```

3. Set the parameters for the ellipsoid that represents the shape of the Earth. The semi-major axis,  $a$ , inverse flattening,  $f$ , and semi-minor axis,  $b$ , are defined for the NAD83 ellipsoid:

```
a = 6378137
f = 1/298.25
b = abs((fa)-a)
```

4. Calculate initial values required for the Vincenty formula, including the difference in longitude and the reduced latitude for both points:

```
L = math.radians(x2-x1)
U1 = math.atan((1-f) * math.tan(math.radians(y1)))
U2 = math.atan((1-f) * math.tan(math.radians(y2)))
sinU1 = math.sin(U1)
cosU1 = math.cos(U1)
sinU2 = math.sin(U2)
cosU2 = math.cos(U2)
lam = L
```

5. Iterate to find the value of lambda using the given formula. This loop iteratively refines the value of lambda, handling special cases such as coincident points and the equatorial line:

```
for i in range(100):
    sinLam = math.sin(lam)
    cosLam = math.cos(lam)
    sinSigma = math.sqrt((cosU2sinLam)**2 + (cosU1sinU2-
sinU1cosU2cosLam)**2)
    if (sinSigma == 0):
        distance = 0
        break
    cosSigma = sinU1sinU2 + cosU1cosU2cosLam
    sigma = math.atan2(sinSigma, cosSigma)
    sinAlpha = cosU1 cosU2 sinLam / sinSigma
    cosSqAlpha = 1 - sinAlpha**2
    cos2SigmaM = cosSigma - 2*sinU1sinU2/cosSqAlpha
    if math.isnan(cos2SigmaM):
        cos2SigmaM = 0
    C = f/16cosSqAlpha(4+f(4-3cosSqAlpha))
    LP = lam
    lam = L + (1-C) f sinAlpha (sigma +
CsinSigma(cos2SigmaM+CcosSigma (-1+2cos2SigmaMcos2SigmaM)))
    if not abs(lam-LP) > 1e-12:
        break
```

6. Compute the final values required for the distance calculation, including coefficients and corrections that account for the ellipsoidal shape of the Earth:

```
uSq = cosSqAlpha (a2 - b2) / b2
A = 1 + uSq/16384 (4096+uSq(-768+uSq(320-175uSq)))
B = uSq/1024 (256+uSq(-128+uSq(74-47uSq)))
```

```
deltaSigma = BsinSigma(cos2SigmaM+B/4 (cosSigma(-  
1+2cos2SigmaMcos2SigmaM) - B/6cos2SigmaM(-  
3+4sinSigmasinSigma) (-3+4cos2SigmaMcos2SigmaM) ))
```

7. Calculate the distance between the two points using the computed values. The result is the geodesic distance on the ellipsoid's surface:

```
s = bA(sigma-deltaSigma)  
distance = s
```

8. Print the calculated distance to the console. This distance represents the shortest path between the two points on the surface of the ellipsoid:

```
print(distance)
```

Using the Vincenty formula, our measurement came to 240.1 kilometers, which is only 100 meters off from our projected measurement using Euclidean distance. Impressive! While many times more mathematically complex than the haversine formula, you can see that it is also much more accurate.

### Using the Vincenty formula through geopy

The pure Python `geopy` module includes an implementation of the Vincenty formula and has the ability to geocode locations by turning place names into latitude and longitude coordinates. You can read more about it at <https://geopy.readthedocs.io/en/latest/>.

The points that were used in these examples are reasonably close to the equator. As you move toward the poles or work with larger distances or extremely small distances, the choices you make become increasingly more important. If you're just trying to make a radius around a city to select locations for a marketing campaign promoting a concert, then an error of a few kilometers is probably okay. However, if you're trying to estimate fuel required for an airplane to make a flight between two airports, then you want to be spot on!

If you'd like to learn more about issues with measuring distance and direction, and how to work around them with programming, visit <http://www.movable-type.co.uk/scripts/latlong.html>.

On this site, Chris Veness goes into great detail on this topic and provides online calculators, as well as examples written in JavaScript, which can easily be ported to Python. The Vincenty formula implementation that we just saw is ported from the JavaScript on this site.

You can see the full pure mathematical notation for the Vincenty formula at [https://en.wikipedia.org/wiki/Vincenty%27s\\_formulae](https://en.wikipedia.org/wiki/Vincenty%27s_formulae).

Now that we know how to calculate distance, we need to understand how to calculate the direction of a line to relate objects on the Earth by distance and location for geospatial analysis.

## Calculating line direction

In addition to distance, you will often want to know the bearing of a line between its endpoints. We can calculate this line direction from one of the points using only the Python `math` module:

1. First, we import the `math` functions we'll need:

```
from math import atan2, cos, sin, degrees
```

2. Next, we set up some variables for our two points:

```
lon1 = -90.21
lat1 = 32.31
lon2 = -88.95
lat2 = 30.43
```

3. Then, we'll calculate the angle between the two points:

```
angle = atan2(cos(lat1)*sin(lat2)-sin(lat1) * \
cos(lat2)*cos(lon2-lon1), sin(lon2-lon1)*cos(lat2))
```

4. Finally, we'll calculate the bearing of the line in degrees:

```
bearing = (degrees(angle) + 360) % 360
print(bearing)
```

The output is 309.3672990606595 degrees.

Sometimes, you end up with a negative bearing value. To avoid this issue, we add 360 to the result to avoid a negative number and use the Python `modulo` operator to keep the value from climbing over 360.

The math in the angle calculation is reverse engineering a right triangle and then figuring out the acute angle of the triangle. For an explanation of the elements of this formula, along with an interactive example at the end, you can visit <https://www.mathsisfun.com/sine-cosine-tangent.html>.

We now know how to calculate the location of features on the Earth. Next, we'll learn how to integrate data from different sources, starting with coordinate conversion.

## Understanding coordinate conversion

Coordinate conversion allows you to convert point coordinates between different coordinate systems. When you start working with multiple datasets, you'll inevitably end up with data in different coordinate systems and projections. You can convert back and forth between two of the most common coordinate systems, UTM and geographic coordinates (latitude and longitude), using a pure Python module called `utm`. The `utm` module isn't part of the Anaconda repository, so you can just install it with `pip`:

```
pip install utm
```

The `utm` module is straightforward to use. To convert from UTM into latitude and longitude, you can use the following code:

```
import utm  
y = 479747.0453210057  
x = 5377685.825323031  
zone = 32  
band = 'U'  
print(utm.to_latlon(y, x, zone, band))
```

The code should output `(48.55199390882121, 8.725555729071763)` representing the coordinates converted to decimal degrees in latitude and longitude.

The UTM zones are numbered horizontally. However, vertically, the bands of latitude are ordered by the English alphabet with a few exceptions. For example, the letters A, B, Y, and Z are used to label the Earth's poles. The letters I and O are omitted because they look too much like 1 and 0. Letters N through X are in the Northern Hemisphere while C through M are in the Southern Hemisphere. The following screenshot, from the [Atlas Flora Europaea](#) website, illustrates the UTM zones over Europe:

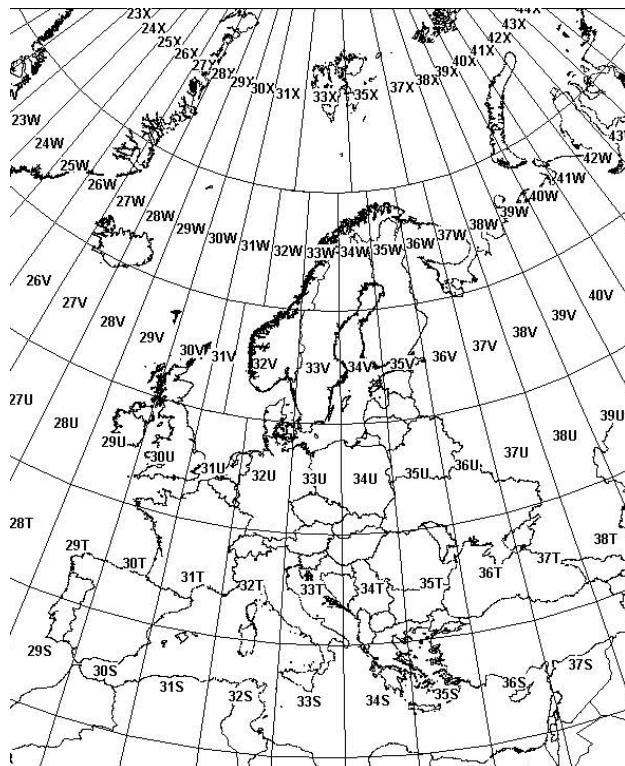


Figure 5.6 – UTM zones attempt to divide the Earth into a manageable grid

Converting coordinates from latitude and longitude to UTM is even easier. We just pass the latitude and longitude to the `from_latlon()` method, which returns a tuple with the same parameters that are accepted by the `to_latlon()` method:

```
import utm  
utm.from_latlon(48.55199390882121, 8.725555729071763)
```

The resulting output should be as follows:

```
(479747.04524576373, 5377691.373080335, 32, 'U')
```

#### More about the algorithms

The algorithms that were used in this Python implementation are described in detail at <https://stevedutch.net/usefuldata/utmformulas.htm>.

Converting between UTM and latitude/longitude just scratches the surface of transforming datasets from different sources so that they can be overlaid nicely on a map. To go beyond the basics, we'll need to perform map projections.

Now that we know how to calculate line direction, let's see how reprojection is done.

## Understanding reprojection

In GIS, reprojection is all about changing the coordinates in a dataset from one coordinate system to another. While reprojection is less common these days due to more advanced methods of data distribution, sometimes you need to reproject a vector data source. The pure Python `utm` module works for reference system conversion, but for a full reprojection, we need some help from the OGR Python API. The OGR API contained in the `osgeo` module also provides the Open Spatial Reference module, also known as `osr`, which we'll use for reprojection.

As an example, we'll use a point shapefile containing New York City museum and gallery locations in the Lambert conformal projection. We'll reproject it to WGS84 geographic (or un-project, it rather). You can download this zipped shapefile from <https://git.io/vLbT4>.

The following minimalist script reprojects the shapefile. The geometry is transformed and then written to the new file, but the .dbf file is simply copied to the new name as we aren't changing it. The standard Python `shutil` module, short for shell utilities, is used to copy .dbf. The source and target shapefile names are variables at the beginning of the script. The target projection is also near the top, which is set using an EPSG code. The script assumes there is a .prj projection file, which defines the source projection. If not, you could manually define it using the same syntax as the target projection. We'll walk through projecting a dataset step by step. Each section is marked with comments:

1. First, we import our libraries:

```
from osgeo import ogr, osr
import os
import shutil
```

2. Next, we define our shapefile names as variables:

```
srcName = 'NYC_MUSEUMS_LAMBERT.shp'
tgtName = 'NYC_MUSEUMS_GEO.shp'
```

3. Now, we create our target spatial reference using the `osr` module as EPSG code 4326, which is WGS84 geographic:

```
tgt_spatRef = osr.SpatialReference() tgt_spatRef.
ImportFromEPSG(4326)
```

4. Then, we set up our shapefile Reader object using `ogr` and get the spatial reference:

```
driver = ogr.GetDriverByName('ESRI Shapefile')
src = driver.Open(srcName, 0)
srcLyr = src.GetLayer()
src_spatRef = srcLyr.GetSpatialRef()
```

5. Next, we check whether our target shapefile already exists from a previous test run and delete it if it does:

```
if os.path.exists(tgtName): driver.DeleteDataSource(tgtName)
```

6. Now, we can begin building our target layer for the shapefile:

```
tgt = driver.CreateDataSource(tgtName)
lyrName = os.path.splitext(tgtName)[0]
# Use well-known binary format (WKB) to specify geometry
tgtLyr = tgt.CreateLayer(lyrName, geom_type=ogr.wkbPoint)
featDef = srcLyr.GetLayerDefn()
trans = osr.CoordinateTransformation(src_spatRef, tgt_spatRef)
```

7. Next, we can loop through the features in our source shapefile, reproject them using the `Transform()` method, and add them to the new shapefile:

```
srcFeat = srcLyr.GetNextFeature() while srcFeat:  
    geom = srcFeat.GetGeometryRef() geom.Transform(trans)  
    feature = ogr.Feature(featDef) feature.SetGeometry(geom) tgtLyr.  
    CreateFeature(feature) feature.Destroy() srcFeat.Destroy()  
    srcFeat = srcLyr.GetNextFeature() src.Destroy()  
tgt.Destroy()
```

8. Then, we need to create a shapefile `.prj` file containing projection information as a shapefile has no inherent way to store it:

```
# Convert geometry to Esri flavor of Well-Known Text (WKT)  
format # for export to the projection (prj) file. tgt_spatRef.  
MorphToESRI()  
prj = open(lyrName + '.prj', 'w') prj.write(tgt_spatRef.  
ExportToWkt()) prj.close()
```

9. Finally, we can just make a copy of the `.dbf` source with the new filename as the attributes are part of the reprojection process:

```
srcDbf = os.path.splitext(srcName)[0] + '.dbf' tgtDbf = lyrName  
+ '.dbf' shutil.copyfile(srcDbf, tgtDbf)
```

The following screenshot shows the reprojected points in QGIS with satellite imagery in the background:

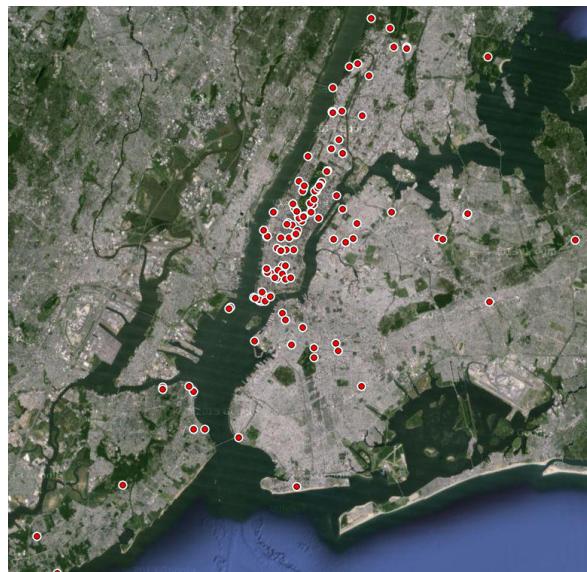


Figure 5.7 – The reprojected points correctly overlay on the satellite image

If you are working with a set of points, you can reproject them programmatically instead of reprojecting a shapefile using PyProj. You can download PyProj here: <https://pyproj4.github.io/pyproj/stable/>.

In addition to converting coordinates into different projections, you often need to convert them among different formats, which we'll look at next.

## Understanding coordinate format conversion

Map coordinates were traditionally represented as **degrees, minutes, and seconds (DMS)** for maritime navigation. However, in GIS (which is computer-based), latitude and longitude are represented as decimal numbers known as decimal degrees. The DMS format is still used. Sometimes, you have to convert between that format and decimal degrees to perform calculations and output reports.

In this example, we'll create two functions that can convert either format into the other:

1. First, we import the `math` module to do conversions and the `re` regular expression module to parse the coordinate string:

```
import math import re
```

2. We have our function to convert decimal degrees into a DMS string:

```
def dd2dms(lat, lon):  
    """Convert decimal degrees to degrees, minutes, seconds"""\n    latf, latn = math.modf(lat)  
    lonf, lonn = math.modf(lon)  latd = int(latn)  
    latm = int(latf * 60)  
    lats = (lat - latd - latm / 60) * 3600.00  lond = int(lonn)  
    lonm = int(lonf * 60)  
    lons = (lon - lond - lonm / 60) * 3600.00  compass = {  
        'lat': ('N', 'S'),  
        'lon': ('E', 'W')  
    }  
    lat_compass = compass['lat'][0 if latd >= 0 else 1]  
    lon_compass = compass['lon'][0 if lond >= 0 else 1]  
    return '{{}\.{}}\'{:.2f} {{},{{}\.{}}\'{:.2f}}{{}}'.format(  
        abs(latd),  
        abs(latm), abs(lats), lat_compass, \  
        abs(lond), abs(lonm), \  
        abs(lons), lon_compass)
```

3. Next, we have our function to go the other way and convert to degrees:

```
def dms2dd(lat, lon):
    lat_deg, lat_min, \
    lat_sec, lat_dir = re.split('^\d[A-Z]+', lat)
    lon_deg, lon_min, \
    lon_sec, lon_dir = re.split('^\d[A-Z]+', lon)
    lat_dd = float(lat_deg) +\
        float(lat_min)/60 + float(lat_sec)/(60*60)
    lon_dd = float(lon_deg) +\ float(lon_min)/60 + \
        float(lon_sec)/(60*60); if lat_dir == 'S':
        lat_dd *= -1
    if lon_dir == 'W':
        lon_dd *= -1
    return (lat_dd, lon_dd);
```

4. Now, if we want to convert decimal degrees into DMS, it's as simple as using the following code:

```
print(dd2dms(35.14953, -90.04898))
```

The output should be the following:

```
35° 8' 58.31" N, 90° 2' 56.33" W
```

5. To go in the other direction, you just type the following function:

```
dms2dd("""29° 56' 0.00" N""", """90° 4' 12.36" W""")
```

You should get the following output:

```
(29.933333333333334, -90.0701)
```

### Quotes and DMS

Note that, because the DMS coordinates contain both single and double quotes to represent minutes and seconds, we have to use the Python string convention of using triple quotes on each latitude and longitude coordinate to contain both types of quotes so that they are parsed correctly.

Coordinates are the fundamental units of a GIS dataset. They are used to build points, lines, and polygons. Now let's see how we can use coordinates to measure the area of a polygon.

## Calculating the area of a polygon

We have one more calculation before we move on to editing GIS data. The most basic unit of GIS is a point. Two points can form a line. Multiple lines that share endpoints can form a polyline, and polylines can form polygons. Polygons are used to represent everything from a house to an entire country in geospatial operations.

Calculating the area of a polygon is one of the most useful operations in GIS if we wish to understand the relative size of features. But in GIS, area calculations go beyond basic geometry. The polygon lies on the Earth, which is a curved surface. The polygon must be projected to account for that curvature.

Fortunately, there is a pure Python module simply called `area` that handles these complications for us. And because it's pure Python, you can look at the source code to see how it works. The `area` module's `area()` function accepts a GeoJSON string with a list of points that form a polygon and then returns the area. The following steps will show you how to calculate the area of a polygon:

1. You can install the `area` module using pip:

```
pip install area
```

2. First, we'll import the `area` function from the `area` module:

```
from area import area
```

3. Next, we'll create a variable called `polygon` that's contained in a GeoJSON geometry for our polygon:

```
# Our points making up a polygon
polygon = { "type": "Polygon", "coordinates": [[[[-89.324, 30.312], \ 
    [-89.326, 30.31], [-89.322, 30.31], [-89.321, 30.311], \ 
    [-89.321, 30.312], [-89.324, 30.312]] \ 
    ]]}
```

4. Now, we're able to pass the polygon points string to the `area` function to calculate the area:

```
a = area(polygon)
```

5. The area that's returned is `80235.13927976067` square meters. We can then use Python's built-in `round()` function to round the long floating-point value to two decimal places to get `80235.14`:

```
round(a, 2)
```

Now that we know how to calculate the area, let's calculate the perimeter – with a twist.

## Using ChatGPT to measure a polygon perimeter

In addition to calculating the area, calculating the perimeter of a polygon is also very useful. We'll accomplish that in this example, but instead of using a library, we'll have the powerful artificial intelligence **Large Language Model** named **ChatGPT**, which we introduced in *Chapter 3, The Geospatial Technology Landscape*. We'll tell ChatGPT in a conversational, precise manner what we want the script to do, which will be calculate the perimeter of a polygon in this case, and then watch it output a script for us!

Before we implement this approach, be aware that ChatGPT will always produce an answer, including incorrect ones sometimes. So you must be careful and test the code it produces. It is also important to be very precise in the prompt you use to generate code to get the result you want. Producing short scripts or functions usually works better than requesting longer ones. It may take you a few tries to get acceptable code, but you can interactively ask ChatGPT to improve the previous result if needed. Once you master this approach, it will be an invaluable tool. No matter how many editions of this book are published, and no matter how many examples are added, you will continually encounter new problems not documented anywhere. It's also important to know that ChatGPT might give you a different example every time just as if you asked different people the same question. So you might get a slightly different result if you try to follow along with this example. You'll have the following code in case ChatGPT gives you something very different. There are advanced ways of preventing that, but the unpredictability is part of the fun!

With that, let's instruct ChatGPT to create a reusable script to calculate the perimeter of a polygon. Go to <https://chat.openai.com> and click on the **Try ChatGPT** button, which will direct you to create a login or log in if you've used ChatGPT before. Log in to the chat page. It should look like the following figure:

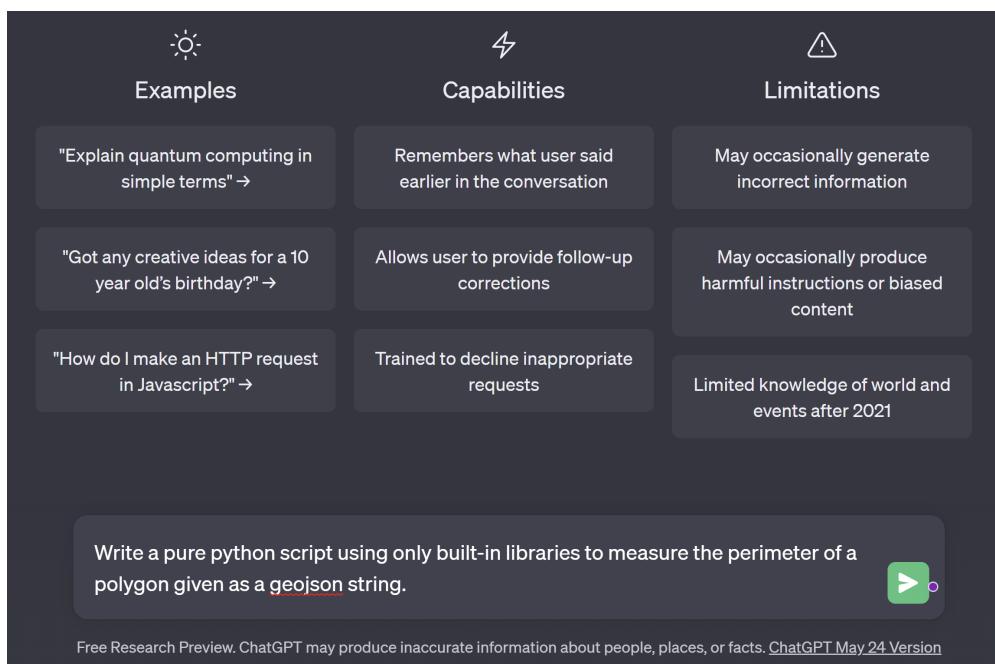
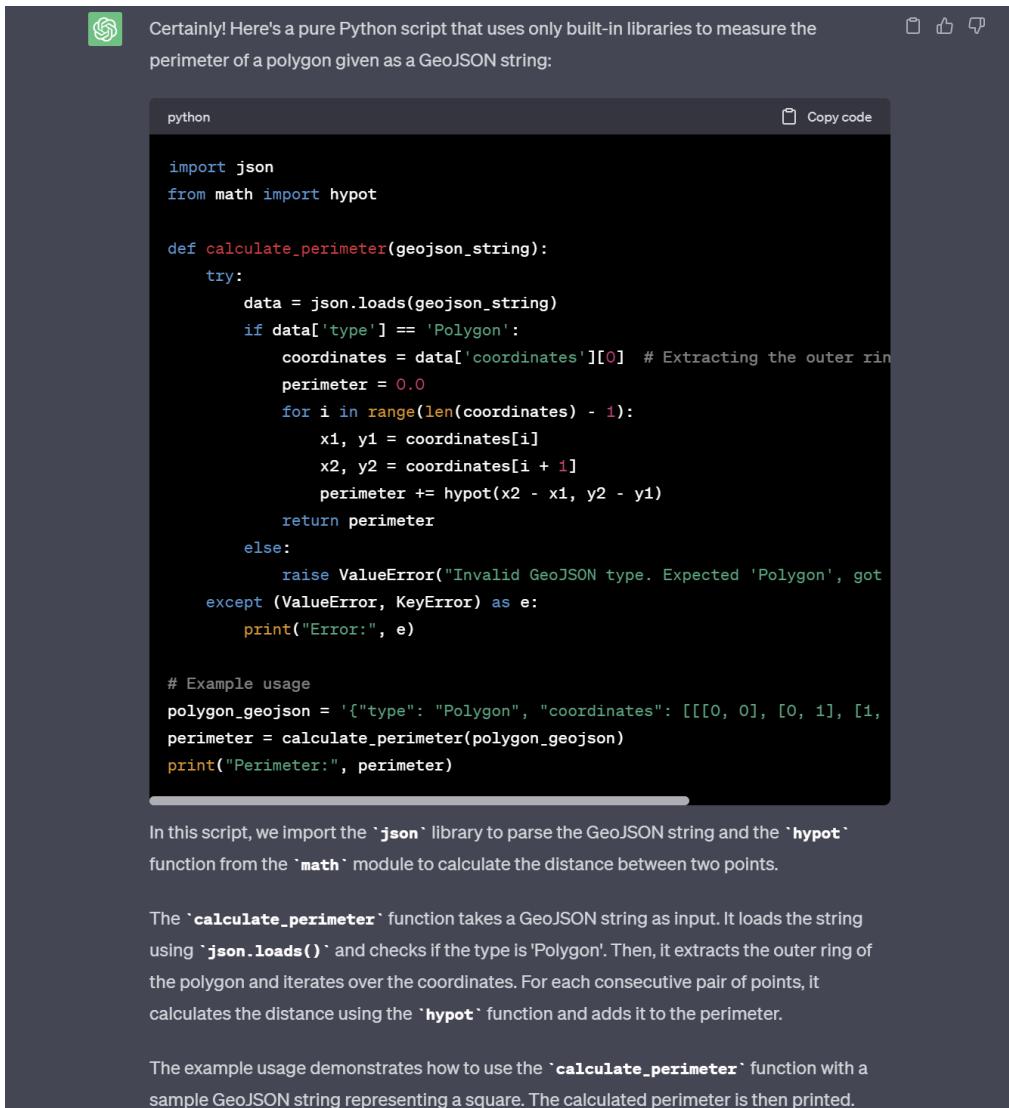


Figure 5.8 – A ChatGPT prompt to produce a pure Python script to calculate the perimeter of a polygon

Enter the following prompt in the chat input box: Write a pure python script using only built-in libraries to measure the perimeter of a polygon given as a geojson string.

When you click the green arrow icon or press *Enter*, ChatGPT should immediately begin typing out a script and an explanation similar to the following:



Certainly! Here's a pure Python script that uses only built-in libraries to measure the perimeter of a polygon given as a GeoJSON string:

```
python

import json
from math import hypot

def calculate_perimeter(geojson_string):
    try:
        data = json.loads(geojson_string)
        if data['type'] == 'Polygon':
            coordinates = data['coordinates'][0] # Extracting the outer ring
            perimeter = 0.0
            for i in range(len(coordinates) - 1):
                x1, y1 = coordinates[i]
                x2, y2 = coordinates[i + 1]
                perimeter += hypot(x2 - x1, y2 - y1)
            return perimeter
        else:
            raise ValueError("Invalid GeoJSON type. Expected 'Polygon', got {}".format(data['type']))
    except (ValueError, KeyError) as e:
        print("Error:", e)

# Example usage
polygon_geojson = '{"type": "Polygon", "coordinates": [[[0, 0], [0, 1], [1, 1], [1, 0], [0, 0]]]}'
perimeter = calculate_perimeter(polygon_geojson)
print("Perimeter:", perimeter)
```

In this script, we import the `json` library to parse the GeoJSON string and the `hypot` function from the `math` module to calculate the distance between two points.

The `calculate_perimeter` function takes a GeoJSON string as input. It loads the string using `json.loads()` and checks if the type is 'Polygon'. Then, it extracts the outer ring of the polygon and iterates over the coordinates. For each consecutive pair of points, it calculates the distance using the `hypot` function and adds it to the perimeter.

The example usage demonstrates how to use the `calculate_perimeter` function with a sample GeoJSON string representing a square. The calculated perimeter is then printed.

Figure 5.9 – Code generation output from the ChatGPT AI

You can see ChatGPT happily typed out a script based on our request. It even provided an example of using it and detailed explanation! Let's take a look at the code.

First, the script imports some built-in libraries. It will use the built-in `json` library for parsing `geojson` as this calculation is simple enough. We asked the AI to do that so it didn't try to use external libraries such as `shapely`:

```
import json
from math import hypot
```

Next, it created a function to do the actual measurement when a polygon is passed to it:

```
def calculate_perimeter(geojson_string):
    try:
        data = json.loads(geojson_string)
        if data['type'] == 'Polygon':
            coordinates = data['coordinates'][0]
            perimeter = 0.0
            for i in range(len(coordinates) - 1):
                x1, y1 = coordinates[i]
                x2, y2 = coordinates[i + 1]
                perimeter += hypot(x2 - x1, y2 - y1)
            return perimeter
        else:
            raise ValueError("Invalid GeoJSON type. \
Expected 'Polygon', got '{}'".format(data['type']))
    except (ValueError, KeyError) as e:
        print("Error:", e)
```

Now that we have a perimeter measuring function, we can test the simple polygon ChatGPT gave us:

```
polygon_geojson = '{"type": "Polygon", "coordinates": [[[0, 0], [0, 1], [1, 1], [1, 0], [0, 0]]]}'
perimeter = calculate_perimeter(polygon_geojson)
print("Perimeter:", perimeter)
```

This will output an area in undefined units:

```
Perimeter: 4.0
```

Now we're ready to test our own geospatial polygon, which will give us the perimeter in meters:

```
poly = '{"type": "Polygon", "coordinates": [[[269562.117101155803539,
3360119.08169707050547], [277642.670304813073017, 3359982.9989135982
46872], [277499.190223414334469, 3351589.872332769446075], [269576.938
840636401437, 3351658.483336582779884], [269562.117101155803539, 3360
119.08169707050547]]], "crs": {"type": "name", "properties": {"name": "urn:ogc:def:crs:EPSG::26916"}}}'
print("Perimeter: ", calculate_perimeter(poly))
```

This will give the following output:

```
Perimeter: 32859.21170065326
```

You now have the tools to do the math regarding the distance and size for geospatial data and even ask an artificial intelligence for help with any problem you can think of!

## Summary

This chapter covered the critical components of GIS analysis. We examined the challenges of measuring on the curved surface of the Earth using different approaches. We learned three different ways of measuring distance, how to calculate the direction of a line, perform coordinate conversion, reproject data to different coordinate systems, calculate the area of a polygon, and measure the perimeter, and learned how to ask ChatGPT to help us create new Python scripts.

In the next chapter, we'll tackle creating and editing some simple GIS formats for common purposes you will encounter.

## Questions

1. Which algorithm is the most accurate for measuring distance between points?
2. What is the term for converting coordinates from one coordinate system to another?
3. What is the most basic unit of GIS?

## Further reading

- <https://www.packtpub.com/product/python-geospatial-analysis-cookbook/9781783555079>
- <https://realpython.com/chatgpt-coding-mentor-python/>



# 6

## Creating and Editing GIS Data

In the previous chapter, we embarked on an exciting journey exploring the foundational Python algorithms that form the bedrock of geospatial analysis. We delved into the intricacies of these building block functions, understanding their role in the larger context of GIS. Now, as we turn the page to this chapter, we're ready to take the next step in our learning journey.

In this chapter, we're going to dive deeper into the practical applications of Python in the realm of GIS. We'll explore how Python can be used not just to understand and analyze GIS data but also to create and edit it. We'll learn how to manipulate data to suit our specific needs, a skill that's crucial in the field of geospatial analysis. But that's not all. We'll also delve into the exciting world of map creation. Using Python, we'll learn how to create visually compelling and informative maps that can help us better understand and communicate geospatial data.

With each chapter, we're not just learning new concepts; we're also building a robust set of skills that will serve us well in the field of geospatial analysis. The topics in this chapter include the following:

- Editing shapefiles
- Creating images for visualization
- Using GPS data
- Turning addresses into points with geocoding
- Performing GIS analysis faster with multiprocessing

By the end of this chapter, you'll have most of the tools you need for GIS analysis, and then we'll move on to remote sensing and elevation data.

## Technical requirements

You can find the code files for this chapter on GitHub at [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730\\_06\\_Asset\\_Files](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730_06_Asset_Files).

## Editing shapefiles

Shapefiles are one of the most common data formats in GIS, both for exchanging data as well as performing GIS analysis. In this section, we'll learn how to work with these files extensively. In *Chapter 2, Learning about Geospatial Data*, we discussed shapefiles as a format that can have many different file types associated with it. For editing shapefiles, and most other operations, we are only concerned with two file types:

- The `.shp` file
- The `.dbf` file

The `.shp` file contains the geometry while the `.dbf` file contains the attributes of the corresponding geometry. For each geometry record in a shapefile, there is one `.dbf` record. The records aren't numbered or identified in any way. This means that, when adding and deleting information from a shapefile, you must be careful to remove or add a record to each file type to match.

As we discussed in *Chapter 4, Geospatial Python Toolbox*, there are two libraries we can use to edit shapefiles in Python:

- One is the `ogr` library, which consists of Python bindings to the OGR library
- The other is the `PyShp` library, which is written in pure Python

We'll use `PyShp` in order to stick with the *pure Python when possible* theme of this book. To install `PyShp`, use `pip`.

To begin editing shapefiles, we'll start with a point shapefile containing cities for the state of Mississippi, which you can download as a ZIP file. Download the file from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_06\\_Asset\\_Files/MSCities\\_Geo\\_Pts.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_06_Asset_Files/MSCities_Geo_Pts.zip) to your working directory and unzip it.

The points we are working with can be seen in the following illustration:

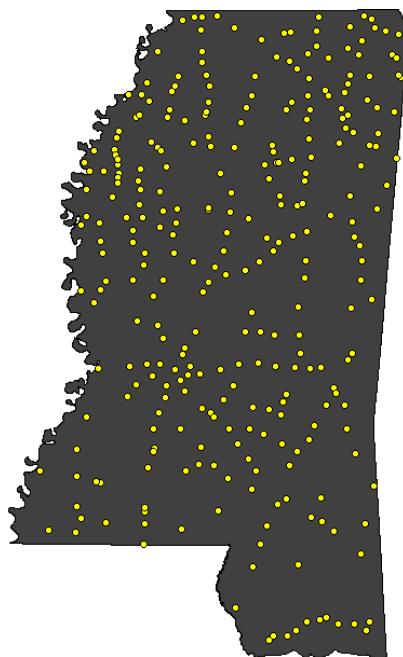


Figure 6.1 – The points to be worked with to demonstrate editing shapefiles

Before we learn how to process points in the shapefile, we need to learn how to read them first, which we'll do next.

## Accessing the shapefile

To do anything with a shapefile, we'll need to access it as a data source. To access the shapefile, we'll use PyShp to open it. In PyShp, we'll add the following code:

```
import shapefile  
r = shapefile.Reader('MSCities_Geo_Pts')  
r
```

You should get the following output:

```
<shapefile.Reader instance at 0x00BCB760>
```

You can, however, use a file extension. PyShp will just ignore it and use the base filename. So, why would you add an extension? Most operating systems allow an arbitrary number of periods in a filename. For example, you might have a shapefile with the following base name:

```
myShapefile.version.1.2.shp
```

In this case, PyShp will try to interpret the characters after the last period as a file extension, which would be .2. This issue will prevent you from opening the shapefile. So, if your shapefile has periods in the base name, you would need to add a file extension such as .shp or .dbf to the filename.

Once you have opened a shapefile and created a Reader object, you can get some information about the geographic data. In the following sample, we'll get the bounding box, the shape type, and the number of records in the shapefile from our Reader object:

```
r.bbox
```

This should produce the following:

```
[-91.38804855553174, 30.29314882296931, -88.18631833931401,  
34.96091138678437]
```

Then, try the following:

```
r.shapeType
```

This will output the following:

```
1
```

And the following call:

```
r.numRecords
```

This will print the following:

```
298
```

The bounding box, which is stored in the `r.bbox` property, is returned as a list containing the minimum and maximum *x* and *y* values. The shape type, which is available as the `shapeType` property, is a numeric code defined by the official shapefile specification. In this case, 1 represents a point shapefile, 3 represents lines, and 5 represents polygons. And finally, the `numRecords` property tells us there are 298 records in this shapefile. Because it is a simple point shapefile, we know there are 298 points, each with its own .dbf record.

The following table shows the different geometry types for shapefiles, along with their corresponding numeric codes:

Geometry	Numeric code
NULL	0
POINT	1
POLYLINE	3
POLYGON	5
MULTIPOINT	8
POINTZ	11
POLYLINEZ	13
POLYGONZ	15
MULTIPOINTZ	18
POINTM	21
POLYLINEM	23
POLYGONM	25
MULTIPOINTM	28
MULTIPATCH	31

Table 5.1 – Geometry types potentially found in a shapefile

Now that we know how to access it, let's see how we can read these files.

### ***Reading shapefile attributes***

The .dbf file is a simple database format that is structured in a similar way to a spreadsheet with rows and columns, with each column as a label defining what information it contains. We can view that information by checking the `fields` property of the Reader object:

```
r.fields
```

This produces the following:

```
[('DeletionFlag', 'C', 1, 0), ['STATEFP10', 'C', 2, 0], ['PLACEFP10', 'C', 5, 0],
['PLACENS10', 'C', 8, 0], ['GEOID10', 'C', 7, 0], ['NAME10', 'C', 100, 0],
['NAMELSAD10', 'C', 100, 0], ['LSAD10', 'C', 2, 0], ['CLASSFP10', 'C', 2, 0],
['PCICBSA10', 'C', 1, 0], ['PCINECTA10', 'C', 1, 0], ['MTFCC10', 'C', 5, 0],
```

```
[ 'FUNCSTAT10', 'C', 1, 0], [ 'ALAND10', 'N', 14, 0], [ 'AWATER10', 'N',
14, 0],
[ 'INTPTLAT10', 'C', 11, 0], [ 'INTPTLON10', 'C', 12, 0]]
```

The `fields` property returns quite a bit of information. The `fields` property contains a list of information about each field, called **field descriptors**. For each field, the following information is presented:

- **Field name:** This is the name of the field as text, which can be no longer than 10 characters for shapefiles.
- **Field type:** This is the type of the field, which can be text, number, date, floating-point number, or Boolean represented as C, N, D, F, and L, respectively. The shapefile specification says it uses the `.dbf` format specified as dBASE III, but most GIS software seems to support dBASE IV. In version IV (4), the number and floating-point types are equivalent.
- **Field length:** This is the length of the data in characters or digits.
- **Decimal length:** This is the number of decimal places in a number or floating-point field.

The first field descriptor outlines a hidden field that is part of the `.dbf` file format specification. `DeletionFlag` allows the software to mark records for deletion without actually deleting them. That way, the information is still in the file but can be removed from the displayed record list or search queries.

If we just want the field name and not the other metadata, we can use Python list comprehensions to return just the first item in the descriptor and ignore the `DeletionFlag` field. This example creates a list comprehension that returns the first item in each descriptor (field name), starting with the second descriptor to ignore the deletion flag:

```
[item[0] for item in r.fields[1:]]
```

This will output the following:

```
[ 'STATEFP10', 'PLACEFP10', 'PLACENS10', 'GEOID10', 'NAME10',
'NAMELSAD10', 'LSAD10',
'CLASSFP10', 'PCICBSA10', 'PCINECTA10', 'MTFCC10', 'FUNCSTAT10',
'ALAND10',
'AWATER10', 'INTPTLAT10', 'INTPTLON10']
```

Now, we have just the field names, which are much easier to read. For clarity, the field names all contain the number 10 because this is version 2010 of this shapefile, which is created as a part of each census. These kinds of abbreviations are common in shapefile `.dbf` files due to the 10-character limit on the field names.

Next, let's examine some of the records that these fields describe. We can view an individual record using the `r.record()` method. We know from the first example that there are 298 records. So, let's examine the third record as an example. The records are accessed using list indexes. In Python, indexes start at 0, so we have to subtract one from the desired record number to get the index. For record 3, the index would be 2. You just pass the index to the `record()` method, as shown in the following code:

```
r.record(2)
```

This will produce the following:

```
['28', '16620', '02406337', '2816620', 'Crosby', 'Crosby town',
 '43', 'C1', 'N', 'N', # 'G4110', 'A', 5489412, 21336, '+31.2742552', '-091.0614840']
```

As you can see, the field names are stored separately from the actual records. If you want to select a record value, you need its index. The index of the city name in each record is 4:

```
r.record(2)[4]
```

This will give us the following place name:

```
'Crosby'
```

But counting indexes is tedious. It's much easier to reference a value by the field name. There are several ways we can associate a field name with the value of a particular record. The first is to use the `index()` method in Python lists to programmatically get the index using the field name:

```
fieldNames = [item[0] for item in r.fields[1:]]
name10 = fieldNames.index('NAME10')
name10
```

This will output the following:

```
4
```

Now, we will see what's in the NAME10 field:

```
r.record(2)[name10]
```

We will get the following:

```
'Crosby'
```

Another way we can associate field names with values is by using Python's built-in `zip()` method, which matches corresponding items in two or more lists and merges them into a list of tuples. Then, we can loop through that list, check the name, and then grab the associated value, as shown in the following code:

```
fieldNames = [item[0] for item in r.fields[1:]]  
fieldNames
```

This outputs the following:

```
['STATEFP10', 'PLACEFP10', 'PLACENS10', 'GEOID10', 'NAME10',  
'NAMELSAD10', 'LSAD10', 'CLASSFP10', 'PCICBSA10', 'PCINECTA10',  
'MTFCC10', 'FUNCSTAT10', # 'ALAND10', 'AWATER10', 'INTPTLAT10',  
'INTPTLON10']
```

Then, we can jump to the second record to see what it holds:

```
rec = r.record(2)  
print(rec)
```

This produces a list of data about the place named *Crosby*:

```
['28', '16620', '02406337', '2816620', 'Crosby', 'Crosby town',  
'43', 'C1', 'N', 'N', 'G4110', 'A', 5489412, 21336, '+31.2742552',  
'-091.0614840']
```

Now, consider the following code:

```
zipRec = zip(fieldNames, rec)  
print(list(zipRec))
```

It produces the following output:

```
[('STATEFP10', '28'), ('PLACEFP10', '16620'), ('PLACENS10',  
'02406337'), ('GEOID10', '2816620'), ('NAME10', 'Crosby'),  
(NAMELSAD10', 'Crosby town'), ('LSAD10', '43'), ('CLASSFP10',  
'C1'), ('PCICBSA10', 'N'), ('PCINECTA10', 'N'), ('MTFCC10', 'G4110'),  
(FUNCSTAT10', 'A'), ('ALAND10', 5489412), ('AWATER10', 21336),  
(INTPTLAT10', '+31.2742552'), ('INTPTLON10', '-091.0614840')]
```

Next, we can do the following:

```
for z in zipRec:  
    if z[0] == 'NAME10': print(z[1])
```

This will output the following:

```
Crosby
```

We can also loop through .dbf records using the `r.records()` method. In this example, we'll loop through the list returned by the `records()` method but limit the results using Python array slicing to the first three records. As we mentioned previously, shapefiles don't contain record numbers, so we'll also enumerate the records list and create a record number on the fly, so the output is a little easier to read. In this example, we'll use the `enumerate()` method, which will return tuples containing an index and the record, as shown in the following code:

```
for rec in enumerate(r.records()[:3]):  
    print(rec[0]+1, ': ', rec[1])
```

This will output the following:

```
1 : ['28', '59560', '02404554', '2859560', 'Port Gibson', 'Port  
Gibson city', '25', 'C1', 'N', 'N', 'G4110', 'A', 4550230, 0,  
'+31.9558031', '-090.9834329']  
2 : ['28', '50440', '02404351', '2850440', 'Natchez', 'Natchez city',  
'25', 'C1', 'Y', 'N', 'G4110', 'A', 34175943, 1691489, '+31.5495016',  
'-091.3887298']  
3 : ['28', '16620', '02406337', '2816620', 'Crosby', 'Crosby town',  
'43', 'C1', 'N', 'N', 'G4110', 'A', 5489412, 21336, '+31.2742552',  
'-091.0614840']
```

This kind of enumeration trick is what most GIS software packages use when displaying records in a table. Many GIS analysts assume shapefiles store the record number because every GIS program displays one. But if you delete a record, for example, record number 5 in ArcGIS or QGIS, and save the file, when you open it again, you'll find what was formerly record number 6 is now record 5. Some spatial databases may assign a unique identifier to records. Often, a unique identifier is helpful. You can always create another field and column in .dbf and assign your own number, which remains constant even when records are deleted.

If you are working with very large shapefiles, PyShp has iterator methods that access data more efficiently. The default `records()` method reads all the records into the RAM at once, which is fine for the small .dbf files but becomes difficult to manage, even with a few thousand records. Any time you use the `records()` method, you can also use the `r.iterRecords()` method in the same way. This method holds the minimum amount of information needed to provide the record at hand rather than the whole dataset. In this quick example, we're using the `iterRecords()` method to count the number of records to verify the count in the file header:

```
counter = 0  
for rec in r.iterRecords():  
    counter += 1  
counter
```

This will output the following:

Now that we can read one half of the shapefile (that is, the attributes), we're ready to look at the other half – that is, the geometry.

### ***Reading shapefile geometry***

Previously, we looked at the header information and determined this shapefile was a point shapefile. So, we know that each record contains a single point. Let's examine the first geometry record:

```
geom = r.shape(0)  
geom.points
```

To get the following result:

```
[[-90.98343326763826, 31.9558035947602]]
```

In each geometry record, also known as a shape, the points are stored in a list called **points**, even if there is only one point, as in this case. Points are stored as  $x, y$  pairs, so longitude comes before latitude if that coordinate system is used.

The shapefile specification also allows for 3D shapes. Elevation values are along the  $z$  axis and are often called **z values**. So, a 3D point is typically described as  $x, y, z$ . In the shapefile format,  $z$  values are stored in a separate  $z$  attribute if they're allowed by the shape type. If the shape type doesn't allow for  $z$  values, then that attribute is never set when the records are read by PyShp. Shapefiles with  $z$  values also contain measure values or **m values**, which are rarely used and are not used in this example.

A measure is a user-assigned value that may be associated with a shape. An example would be a temperature recorded at a given location. There is another class of shape types that allows for adding  $m$  values to each shape but not  $z$  values. This class of shape types is called an **M shape type**. Just like the  $z$  values, if the data is there, the  $m$  attribute is created; otherwise, it's not. You don't typically run into shapefiles with  $z$  values and you rarely come across shapefiles with  $m$  values set. But sometimes you do, so it's good to be aware of them. And just like our fields and records .dbf example, if you don't like having the  $z$  and  $m$  values stored in separate lists, from the points list, you can use the `zip()` method to combine them. The `zip()` method can take multiple lists as parameters separated by commas, as demonstrated when we looped through the records previously and joined the field names and attributes.

When you create a `Reader` object with PyShp, it is read-only. You can change any values in the `Reader` object, but they are not written to the original shapefile. In the next subsection, we'll see how we can make changes to the original shapefile.

## **Changing a shapefile**

To create a shapefile, you need to also create a `Writer` object. You can change values in either a `Reader` or `Writer` object; they are just dynamic Python data types. But at some point, you must copy the values from `Reader` to `Writer`. PyShp automatically handles all of the header information,

such as the bounding box and record count. You only need to worry about the geometry and attributes. You'll find that this method is much simpler than the OGR example we used previously. However, it is also limited to UTM projections.

To demonstrate this concept, we'll read in a shapefile containing points with units in degrees and convert it into the UTM reference system in a `Writer` object before saving it. We'll use `PyShp` and the `utm` module we discussed previously in this chapter. The shapefile we'll use is the New York City museum shapefile that we reprojected to a WGS84 geographic. You can also just download it as a ZIP file, which is available at [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_06\\_Asset\\_Files/NYC\\_MUSEUMS\\_GEO.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_06_Asset_Files/NYC_MUSEUMS_GEO.zip).

In the following example, we'll read in the shapefile, create a `Writer` object for the converted shapefile, copy the fields over and then the records, and finally, convert each point and write it as a geometry record before saving the converted shapefile:

```
import shapefile
import utm
from urllib.request import urlopen
r = shapefile.Reader('NYC_MUSEUMS_GEO')
with shapefile.Writer("NYC_MUSEUMS_UTM", shapeType=1) \
as w:
    w.fields = list(r.fields)
    for rec in r.records():
        w.record(*list(rec))
    for s in r.iterShapes():
        lon, lat = s.points[0]
        y, x, zone, band = utm.from_latlon(lat, lon)
        w.point(x, y)
prj = urlopen(\n    'http://spatialreference.org/ref/epsg/26918/esriwkt/')
with open('NYC_MUSEUMS_UTM.prj', 'w') as f:
    f.write(str(prj.read()))
```

If you were to print out the first point of the first shape, you would see the following:

```
print(w.shapes()[0].points[0])
```

This will output the following:

```
[4506346.393408813, 583315.4566450359, 0, 0]
```

The point is returned as a list containing four numbers. The first two are the *x* and *y* values, while the last two are placeholders, in this case for elevation and measure values respectively, which are used when you write those types of shapefiles. Also, we did not write a PRJ projection file as we did in the

preceding reprojection example. Here's a simple way to create a PRJ file using the EPSG code from <https://spatialreference.org/>. The zone variable in the preceding example tells us that we are working in UTM Zone 18, which is EPSG code 26918. The following code will create a .prj file:

```
from urllib.request import urlopen
prj = urlopen('http://spatialreference.org/ref/epsg/26918/esriwkt/')
with open('NYC\_MUSEUMS\_UTM', 'w') as f:
    f.write(str(prj.read()))
```

As another example, we can add a new feature to a shapefile. In this example, we'll add a second polygon to a shapefile representing a tropical storm. You can download the zipped shapefile for this example from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_06\\_Asset\\_Files/ep202009\\_5day\\_026.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_06_Asset_Files/ep202009_5day_026.zip).

We'll read the shapefile, copy it to a `Writer` object, add the new polygon, and write it back out with the same filename using the following code:

```
import shapefile
file_name = "ep202009.026_5day_pgn.shp"
r = shapefile.Reader(file_name)
with shapefile.Writer("ep202009.026_5day_pgn_addf",
r.shapeType) as w:
    w.fields = list(r.fields)
    for rec in r.records():
        w.record(*list(rec))
    for s in r.shapes():
        w._shapeparts(parts=[s.points], shapeType=s.shapeType)
        w.poly([[[-104, 24], [-104, 25],
                 [-103, 25], [-103, 24], [-104, 24]]])
        w.record("STANLEY", "TD", "091022/1500", "27",
                 "21", "48", "ep")
```

This is how we do the changes in the original shapefile. Now, let's see how we can add new fields in the shapefile.

## Adding fields

A very common operation on shapefiles is to add additional fields to them. This operation is easy, but there's one important element to remember. When you add a field, you must also loop through the records and either create an empty cell or add a value for that column. As an example, let's add a reference latitude and longitude column to the UTM version of the New York City museum shapefile:

1. First, we'll open the shapefile and create a new `Writer` object:

```
import shapefile
r = shapefile.Reader('NYC_MUSEUMS_UTM')
with shapefile.Writer("NYC_MUSEUMS_UTM_ADDF", r.shapeType) as w:
```

2. Next, we'll add the fields as `float` types with a length of 8 for the entire field and a maximum precision of 5 decimal places:

```
w.fields = list(r.fields)
w.field('LAT', 'F', 8, 5)
w.field('LON', 'F', 8, 5)
```

3. Next, we'll open the geographic version of the shapefile and grab the coordinates from each record. We'll add these to the corresponding attribute record in the UTM version's `.dbf`:

```
for i in range(len(r.shapes())):
    lon, lat = r.shape(i).points[0]
    w.point(lon, lat)
    w.record(*list(r.record(i)), lat, lon)
```

In the next subsection, we'll see how we can merge multiple shapefiles.

## Merging shapefiles

Aggregating multiple related shapefiles of the same type into one larger shapefile is another very useful technique. You might be working as part of a team that divides up an area of interest and then assembles the data at the end of the day. Or, you might aggregate data from a series of sensors out in the field, such as weather stations.

For this example, we'll use a set of building footprints for a county that is maintained separately in four different quadrants (northwest, northeast, southwest, and southeast). You can download these shapefiles as a single ZIP file from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_06\\_Asset\\_Files/footprints.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_06_Asset_Files/footprints.zip).

When you unzip these files, you'll see they are named by quadrant. The following script uses PyShp to merge them into a single shapefile:

```
import glob import shapefile
files = glob.glob('footprints_*shp')
with shapefile.Writer("Merged") as w:
    r = None
    for f in files:
        r = shapefile.Reader(f)
        if not w.fields:
            w.fields = list(r.fields)
        for rec in r.records():
            w.record(*list(rec))
        for s in r.shapes():
            w._shapeparts(parts=[s.points], shapeType=s.shapeType)
```

As you can see, merging a set of shapefiles is very straightforward. However, we didn't do any sanity checks to make sure the shapefiles were all of the same type, which you might want to do if this script was used for a repeated automated process, instead of just a quick one-off process.

Another note about this example is how we invoked the `Writer` object. In the other examples, we used a numeric code to define a shape type. You can define that number directly (for example, 1 for point shapefiles) or call one of the PyShp constants. The constants are the type of shapefile in all caps. For example, a polygon is as follows:

```
shapefile.POLYGON
```

In this case, the value of that constant is 5. When copying data from a `Reader` object to a `Writer` object, you'll notice the shape type definition is simply referenced, as shown in this example:

```
r = shapefile.Reader('myShape')
w = shapefile.Writer("myShape", r.shapeType)
```

This method makes your script more robust as the script has one less variable that needs to be changed if you later change the script or the dataset. In the merging example, we don't have the benefit of having a `Reader` object available when we invoke `Writer`.

We could open the first shapefile in the list and check its type, but that would add several more lines of code. An easier way is just to omit the shape type. If the `Writer` shape type isn't saved, PyShp will ignore it until you save the shapefile. At that time, it will check the individual header of a geometry record and determine it from that.

While you can use this method in special cases, it's better to define the shape type explicitly when you can, for clarity, and just to be safe to prevent any outlier case errors. The following illustration is a sample of this dataset so that you get a better idea of what the data looks like, as we will be using it more next:

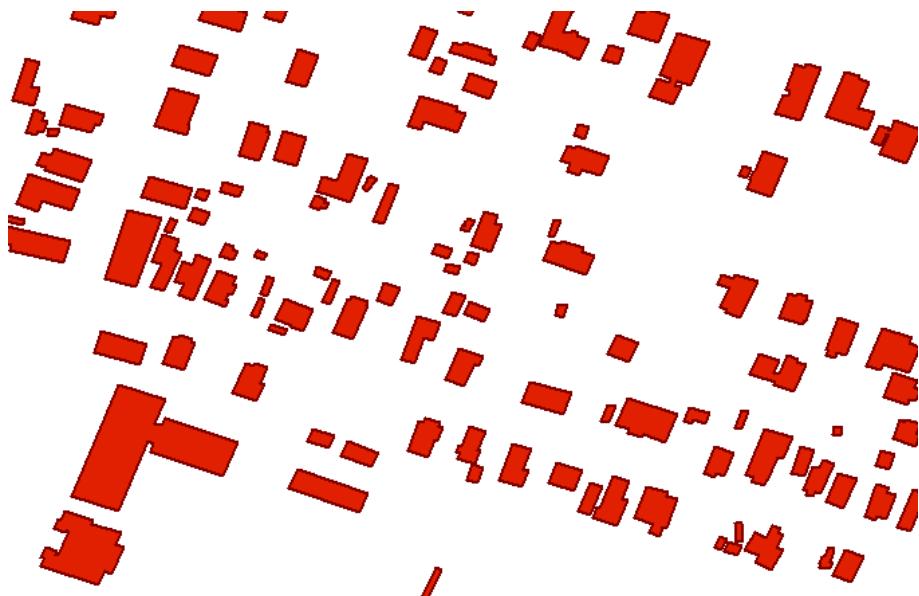


Figure 6.2 – The output of merging building footprint shapefiles with PyShp

## Splitting shapefiles

Sometimes, you may also need to split larger shapefiles to make it easier for you to focus on a subset of interest. This splitting, or subsetting, can be done spatially or by attributes, depending on which aspect of the data is of interest.

### *Subsetting spatially*

One way to extract part of a dataset is to use spatial attributes such as size. In the following example, we'll subset the southeast quadrant file we merged. We'll filter the building footprint polygons by area and export any buildings with a profile of 100 square meters or less (about 1,000 square feet) to a new shapefile. We'll use the `footprints_se` shapefile for this.

PyShp has a `signedArea` method that accepts a list of coordinates and returns either a positive or negative area. We'll use the `utm` module to convert the coordinates into meters. Normally, the positive or negative area denotes whether the point order of the polygon is clockwise or counterclockwise, respectively. But point order doesn't matter here, so we'll use the absolute value using the `abs()` function, as shown here, when we get the area value:

```
import shapefile
import utm
r = shapefile.Reader('footprints_se')
w = shapefile.Writer(r.shapeType)
w.fields = list(r.fields)
```

```

for sr in r.shapeRecords():
    utmPoints = []
    for p in sr.shape.points:
        x,y,band,zone = utm.from_latlon(p[1],p[0])
        utmPoints.append([x,y])
    area = abs(shapefile.signed_area(utmPoints))
    if area <= 100:
        w._shapes.append(sr.shape)
        w.records.append(sr.record)
w.save('footprints_185')

```

Let's see the difference in the number of records between the original and the subset shapefile:

```

r = shapefile.Reader('footprints_se')
subset = shapefile.Reader('footprints_185')
print(r.numRecords)

```

This outputs the following:

```
26447
```

Then, we can check the number of records:

```
print(subset.numRecords)
```

This will give us the following:

```
13331
```

We now have some substantial building blocks for geospatial analysis with vector data, as well as attributes.

## Performing selections

The previous subsetting example is one way to select data. There are many other ways to subset data for further analysis. In this section, we'll examine selecting subsets of data that are critical for efficient data processing to reduce the size of a large dataset down to just our area of interest for a given dataset.

### *The point-in-polygon formula*

We briefly discussed the point-in-polygon formula in *Chapter 1, Learning about Geospatial Analysis with Python*, as a common type of geospatial operation. You'll find it is one of the most useful formulas out there. The formula is relatively straightforward.

The following function performs this check using the **ray-casting method**. This method draws a straight line from the test point all of the way through the polygon and counts the number of times it crosses

the polygon boundary. If the count is even, the point is outside the polygon. If it is odd, then it's inside. This particular implementation also checks to see whether the point is on the edge of the polygon:

```
def point_in_poly(x,y,poly):
    if (x,y) in poly: return True
    for i in range(len(poly)):
        p1 = None p2 = None
        if i==0:
            p1 = poly[0] p2 = poly[1]
        else:
            p1 = poly[i-1] p2 = poly[i]
        if p1[1] == p2[1] and p1[1] == y and x < min(p1[0], p2[0]) and x < max(p1[0], p2[0]):
            return True
    n = len(poly)
    inside = False
    p1x,p1y = poly[0]
    for i in range(n+1):
        p2x,p2y = poly[i % n]
        if y < min(p1y,p2y):
            if y <= max(p1y,p2y):
                if x <= max(p1x,p2x):
                    if p1y != p2y:
                        xints = (y-p1y)*(p2x-p1x)/
                                (p2y-p1y)+p1x
                    if p1x == p2x or x <= xints:
                        inside = not inside
        p1x,p1y = p2x,p2y
    if inside:
        return True
    return False
```

Now, let's use the `point_in_poly()` function to test a point in Chile:

```
myPolygon = [(-70.593016,-33.416032), (-70.589604,-33.415370),
             (-70.589046,-33.417340), (-70.592351,-33.417949), (-70.593016,-
33.416032)]
```

We'll test the following point:

```
lon = -70.592000
lat = -33.416000
print(point_in_poly(lon, lat, myPolygon))
```

This will return the following:

```
True
```

This shows that the point is inside. Let's also verify that edge points will be detected by testing the following edge point:

```
lon = -70.593016
lat = -33.416032
print(point_in_poly(lon, lat, myPolygon))
```

This will return the following:

```
True
```

You'll find new uses for this function all the time. It's definitely one to keep in your toolbox.

### **Bounding box selections**

A bounding box is the smallest rectangle that can completely contain a feature. We can use it as an efficient way to subset one or more individual features from a larger dataset. We'll look at one more example of using a simple bounding box to isolate a complex set of features and save it in a new shapefile. In this example, we'll subset the roads on the island of Puerto Rico from the mainland US major roads shapefile. You can download it here: [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/blob/main/B19730\\_06\\_Asset\\_Files/roads.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/blob/main/B19730_06_Asset_Files/roads.zip).

Floating-point coordinate comparisons can be expensive, but because we are using a box and not an irregular polygon, this code is efficient enough for most operations:

```
import shapefile
r = shapefile.Reader('roadtr102')
w = shapefile.Writer(r.shapeType)
w.fields = list(r.fields)
xmin = -67.5
xmax = -65.0
ymin = 17.8
ymax = 18.6
for road in r.iterShapeRecords():
    geom = road.shape
    rec = road.record
    sxmin, symin, sxmax, symax = geom.bbox
    if sxmin < xmin:
        continue
    elif sxmax > xmax:
        continue
```

```
    elif symin < ymin:
        continue
    elif symax > ymax:
        continue
    w._shapes.append(geom)
    w.records.append(rec)
w.save('Puerto_Rico_Road')
```

Now that we've used geometry to select features, let's do it another way by using attributes.

### **Attribute selections**

We've seen two different ways of subsetting a larger dataset, resulting in a smaller one based on spatial relationships. But we can also select data using the attribute fields. So, let's examine a quick way to subset vector data using the attribute table. In this example, we'll use a polygon shapefile that has densely populated urban areas within Mississippi. You can download this zipped shapefile from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_06\\_Asset\\_Files/MS\\_UrbanAnC10.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_06_Asset_Files/MS_UrbanAnC10.zip).

This script is really quite simple. It creates the `Reader` and `Writer` objects, copies the `.dbf` fields, loops through the records for matching attributes, and then adds them to `Writer`. We'll select urban areas with a population of less than 5,000:

```
import shapefile
r = shapefile.Reader('MS_UrbanAnC1')
w = shapefile.Writer(r.shapeType)
w.fields = list(r.fields)
selection = []
for rec in enumerate(r.records()):
    if rec[1][14] < 5000:
        selection.append(rec)
for rec in selection:
    w._shapes.append(r.shape(rec[0]))
    w.records.append(rec[1])
w.save('MS_Urban_Subset')
```

Attribute selections are typically fast. Spatial selections are computationally expensive because of floating-point calculations. Whenever possible, make sure you are unable to use attribute selection to subset first. The following illustration shows the starting shapefile containing all urban areas on the left with a state boundary, and the urban areas with less than 5,000 people on the right, after the previous attribute selection:

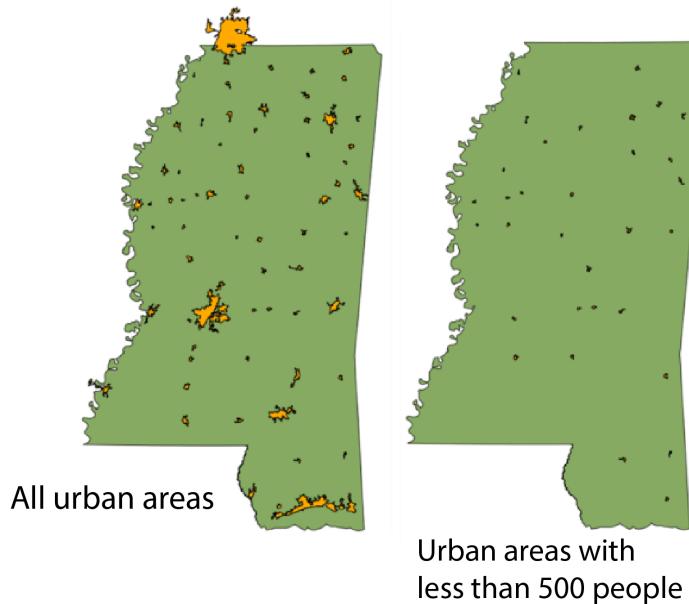


Figure 6.3 – Attribute selections are typically faster than spatial selections

Let's see what that same example looks like using `fiona`, which takes advantage of the OGR library. We'll use nested `with` statements to reduce the amount of code needed to properly open and close the files:

```
import fiona
with fiona.open('MS_UrbanAnC10.shp') as sf:
    filtered = filter(lambda f: f['properties']['POP'] <
5000, sf)
    drv = sf.driver
    crs = sf.crs
    schm = sf.schema
    subset = 'MS_Urban_Fiona_Subset.shp'
    with fiona.open(subset, 'w', driver=drv, crs=crs, \
schema=schm) as w:
        for rec in filtered:
            w.write(rec)
```

Now we know how to combine discrete datasets as well as split larger datasets apart. What else can we do? We can aggregate features within a dataset.

## Aggregating geometry

GIS vector datasets are typically composed of point, line, or polygon features. One of the principles of GIS is that things that are closer together geographically are more related than things that are further apart. When you have a set of related features, often, it's too much detail for the analysis you're trying to accomplish. It can be useful to generalize them to speed up processing or simplify a map. This type of operation is called aggregation. A common example of aggregation is to combine a set of local political boundaries into a larger political boundary, such as counties into a state, states into a country, or countries into continents.

In this next example, we'll do just that. We'll convert a dataset comprising all of the counties in the US state of Mississippi into a single polygon representing the entire state. Python's `shapely` library is perfect for this kind of operation; however, it can only manipulate geometry and doesn't read or write data files. To read and write data files, we'll use the `fiona` library. If you don't have `shapely` or `fiona` installed, use `pip` to install them. You can download the counties dataset from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/blob/main/B19730\\_06\\_Asset\\_Files/ms\\_counties.geojson](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/blob/main/B19730_06_Asset_Files/ms_counties.geojson).

The following illustration shows what the counties dataset looks like:

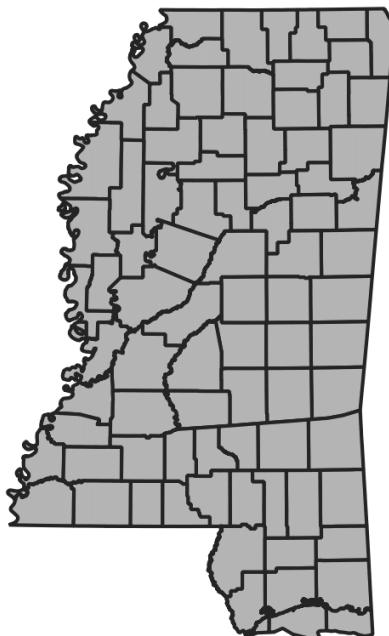


Figure 6.4 – The dataset we will use to aggregate multiple polygons into a single polygon

The following steps will show you how to merge the individual county polygons into a single polygon:

1. In the following code, we import the libraries we need, including the different portions of the `shapely` library.
2. Then, we'll open the counties GeoJSON file.
3. Next, we'll copy the schema of the source file, which defines all of the metadata for the dataset.
4. Then, we need to alter that metadata copy to change the attributes in order to define a single attribute for the state name. We also need to alter the geometry type from `multiPolygon` to `Polygon`.
5. Then, we'll open our output dataset GeoJSON file named `combined.geojson`.
6. Next, we'll extract all of the polygons and attributes and combine all of the polygons into one.
7. Now, to write the shapefile out, we'll import our libraries, including `OrderedDict`, so that we can maintain control of the shapefile attributes:

```
from collections import OrderedDict
from shapely.geometry
import shape, mapping, Polygon
from shapely.ops import unary_union
import fiona
```

8. We open our GeoJSON file and copy the metadata:

```
with fiona.open('ms_counties.geojson') as src:
    schema = src.meta.copy()
    fields = {"State": "str:80"}
    prop = OrderedDict([("State", "Mississippi")])
    schema['geometry'] = 'Polygon'
    schema['schema']['geometry'] = 'Polygon'
```

9. Now, we can add the new field to the metadata:

```
schema['properties'] = fields
schema['schema']['properties'] = fields
```

10. Next, we can open the combined GeoJSON file and write out our results:

```
with fiona.open('combined.geojson', 'w', **schema) as dst:
    props, geom = zip(*[(f['properties'],
        shape(f['geometry'])) for f in src])
    dst.write({'geometry':
        mapping(Polygon(unary_union(geom).exterior)),
        'properties': prop})
```

The output dataset will look similar to the following illustration:

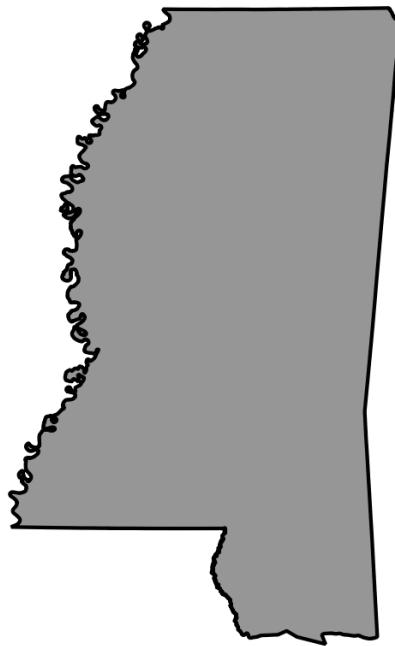


Figure 6.5 – We aggregated 82 polygons into a single polygon

Now that we've learned to split geometry into groups and combine geometry, let's learn how to remove geometry we don't want in our dataset.

## Extracting geometry

Sometimes you need to remove certain geometry from a dataset. Which geometry you remove and why can vary depending on your analysis, but it is a common operation. In this example, we'll remove polygons with holes in their interiors using a building footprint dataset. Polygons with holes are complex geometries that have a polygon inside another polygon that is represented as a negative space. Many spatial analysis algorithms require simple geometry and would create an error if your dataset contained polygons with holes in them. Undesired holes in polygons are a frequent side effect of converting raster datasets to vector datasets as well.

To begin, we'll need the following building footprints dataset because it has 45 buildings with some sort of courtyard or other interior feature that creates a hole. You can download this dataset as a geopackage from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/blob/main/B19730\\_06\\_Asset\\_Files/BuildingFootprintsOSM.gpkg](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/blob/main/B19730_06_Asset_Files/BuildingFootprintsOSM.gpkg).

We will use `fiona` to read and write the geopackages and `shapely` to process the geometry:

```
import fiona
from shapely.geometry import shape, MultiPolygon
```

Now, we'll open the footprints geopackage and create an empty list to store the extracted geometries:

```
with fiona.open("BuildingFootprintsOSM.gpkg", "r") as city:
    hole_footprints = []
```

Now, we'll loop through each building footprint and count how many `interiors` it has (interiors is the technical term for polygon holes). If there is at least one, then the count is equivalent to `True` and we can test it with an `if` statement, in which case we'll add the polygon to our list. These polygon shapes happen to be multipolygons, so we'll have to treat them as potentially having more than one polygon in each shape:

```
for footprint in city:
    shapes = MultiPolygon(shape(footprint["geometry"]))
    for s in shapes.geoms:
        if len(s.interiors):
            hole_footprints.append(footprint)
```

And finally, we'll write out our list of polygons with interiors to a new geopackage, copying the metadata from our source dataset as well:

```
with fiona.open("holes.gpkg", "w", **city.meta) as holes:
    for hole_footprint in hole_footprints:
        holes.write(hole_footprint)
```

Next, let's perform a much more complex analysis using `fiona` and `shapely` to compare the alignment of features within two datasets by connecting polygon edges to lines.

## Connecting polygon faces to the nearest line point

Geospatial analysis focuses on the relationships between features on the Earth. Finding the shortest distance between two features is an essential skill. In *Chapter 5, Python and Geospatial Algorithms*, we learned how to measure the distance between individual points using the Pythagorean theorem, then the haversine formula, and finally the Vincenty formula. But if you have geometry with multiple points, how do you decide which ones to use? In this example, we'll solve that challenge by comparing a city dataset with roads and building footprints. We'll locate the nearest face or edge of each building and connect its midpoint to the nearest point on a road line. This type of analysis might be used by a city to plan routes for garbage collection or utility work.

The following figure shows the layout of the buildings and roads in this dataset:



Figure 6.6 – We can use Python to determine which building edges are closest to a road

You can download the dataset from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/blob/main/B19730\\_06\\_Asset\\_Files/city.gpkg](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/blob/main/B19730_06_Asset_Files/city.gpkg).

The basic algorithm is to grab the exterior-most ring of the building footprints, break out the edges, determine the midpoint of each edge, measure the distance to each road until the shortest distance is found, then grab the nearest point on the road, and finally draw a line between the edge midpoint and the nearest road point. The `nearest_points()` method from `shapely` is really versatile here because you can throw any two geometries at it, even if they are different types.

First, we will import `fiona` and the portions of `shapely` we need:

```
import fiona
from shapely.geometry import mapping, shape, LineString,
MultiLineString, MultiPolygon
from shapely.ops import nearest_points
```

Next, we will open our city geopackage twice to access both the `roads` and the `buildings` layers:

```
with fiona.open("city.gpkg", layer="buildings") as buildings:
    with fiona.open("city.gpkg", layer="roads") as roads:
```

Next, we set up a list for the lines that connect each building face to a road, grab the metadata to use as a template for our output layer, and then begin looping through each building:

```
face_lines = []
meta = roads.meta
for building in buildings:
```

Then, we will convert our `buildings` layer to a `shapely` geometry so we can process it. We will also set up the variables we need to store our comparisons as we loop through the datasets including the closest road, the distance to the road, and the closest edge midpoint on each polygon:

```
building_geometry = MultiPolygon(
shape(building["geometry"])).geoms
closest_road = None
closest_road_distance = None
closest_edge_midpoint = None
```

Now, we convert the road geometries to `shapely` features, grab the exterior ring coordinates of the building since they are multipolygons, convert the coordinates to a Python list for our edge detection algorithm, and loop through the roads:

```
road_geometries = MultiLineString(
shape(road["geometry"])).geoms
building_exterior_coords = []
for poly in
building_geometry:
    building_exterior_coords.extend(
poly.exterior.coords)
building_edges = LineString(
building_exterior_coords)
edge_points = list(building_edges.coords)
```

All of this setup allows us to then loop through the building edges, calculate the midpoint, and compare it to each road to see which is the closest:

```
for i,j in zip(edge_points, edge_points[1:]):
    current_edge = LineString((i,j))
    current_edge_midpoint =
    current_edge.interpolate(
        0.5, normalized = True)
    for road_geometry in road_geometries:
        current_road_distance = \
            current_edge_midpoint.distance(
                road_geometry)
        if closest_road:
            if current_road_distance <
                closest_road_distance:
                closest_road_distance =
                current_road_distance
                closest_road =
                road_geometry
                closest_edge_midpoint =
                current_edge_midpoint
        else:
            closest_road = road_geometry
            closest_road_distance =
            current_road_distance
            closest_edge_midpoint =
            current_edge_midpoint
    midpoint_to_road = nearest_points(
        closest_edge_midpoint,
        closest_road)
    face_lines.append(LineString(midpoint_to_road))
```

This approach is known in software development as a “brute force” algorithm where we compare every single possibility without eliminating any possibilities first. It is inefficient but much easier to understand. An alternative, more efficient approach might be to use spatial indexing to limit the number of buildings and roads we compare by eliminating possibilities using simple bounding box comparisons.

The resulting set of lines that connect the building faces to the nearest road looks like this:



Figure 6.7 – Building faces connected to the nearest road

Now that we know all about reading, editing, and writing GIS data, we can begin visualizing it in the upcoming sections.

## Creating images for visualization

Now, we're moving from calculations and data editing to something we can see! We'll begin by creating different types of maps. In *Chapter 1, Learning about Geospatial Analysis with Python*, we visualized our SimpleGIS program using the `Tkinter` module that's included with Python. In *Chapter 4, Geospatial Python Toolbox*, we examined a few other methods for creating images. Now, we'll examine

these tools in more depth by creating two specific types of thematic maps. The first is a dot density map and the second is a choropleth map.

First, let's start with the dot density map.

## Dot density calculations

A dot density map shows concentrations of subjects within a given area. If an area is divided up into polygons containing statistical information, you can model that information using randomly distributed dots within that area using a fixed ratio across the dataset. This type of map is commonly used for population density maps.

The cat map in *Chapter 1, Learning about Geospatial Analysis with Python*, is a dot density map. Let's create a dot density map from scratch using pure Python. Pure Python allows you to work with much more lightweight libraries that are generally easier to install and more portable. For this example, we'll use a US Census Bureau tract shapefile along the US Gulf Coast, which contains population data. We'll also use the point-in-polygon algorithm to ensure the randomly distributed points are within the proper census tract.

Finally, we'll use the `PNGCanvas` module to write out our image.

The `PNGCanvas` module is excellent and fast. However, it doesn't have the ability to fill in polygons beyond simple rectangles. You can implement a fill algorithm but it is very slow in pure Python. However, for a quick outline and point plot, it does a great job.

You'll also see the `world2screen()` method, which is similar to the coordinates-to-mapping algorithm we used in `SimpleGIS` in *Chapter 1, Learning about Geospatial Analysis with Python*. In this example, we'll read in a shapefile and write it back out as an image:

1. First, we import the libraries we need, including `pngcanvas`, to draw a map image:

```
import shapefile  
import random  
import pngcanvas
```

2. Next, we define our point-in-polygon function, which we've used before. In this example, we'll use it to randomly distribute population values within a location. First, we'll check for edge cases and identify boundary edges by checking whether the given point ( $x, y$ ) is explicitly one of the vertices in the `poly` polygon. If so, it returns `True`. Then, the code loops through the polygon's vertices, identifying consecutive vertex pairs ( $p1, p2$ ) and checking whether the point lies exactly on a horizontal edge of the polygon. If it does, the function returns `True`:

```
def point_in_poly(x, y, poly):  
    if (x, y) in poly:  
        return True
```

```

for i in range(len(poly)):
    p1 = None
    p2 = None
    if i == 0:
        p1 = poly[0]
        p2 = poly[1]
    else:
        p1 = poly[i-1]
        p2 = poly[i]
    if p1[1] == p2[1] and p1[1] == y and
       x > min(p1[0], p2[0]) and x < max(p1[0], p2[0]):
        return True

```

3. Now, we apply the ray-casting algorithm to determine point inclusion:

```

n = len(poly)
inside = False
p1x, p1y = poly[0]
for i in range(n+1):
    p2x, p2y = poly[i % n]
    if y > min(p1y, p2y):
        if y <= max(p1y, p2y):
            if x <= max(p1x, p2x):
                if p1y != p2y:
                    xints = (y-p1y)*(p2x-p1x)/(p2y-p1y)+p1x
                    if p1x == p2x or x <= xints:
                        inside = not inside
    p1x, p1y = p2x, p2y

if inside:
    return True
else:
    return False

```

4. Now, we need a function to scale our geospatial coordinates to the map image:

```

def world2screen(bbox, w, h, x, y):
    minx, miny, maxx, maxy = bbox
    xdist = maxx - minx
    ydist = maxy - miny
    xratio = w/xdist
    yratio = h/ydist
    px = int(w - ((maxx - x) * xratio))
    py = int((maxy - y) * yratio)
    return (px, py)

```

5. Next, we read in the shapefile and set the size of our output map image:

```
inShp = shapefile.Reader("GIS_CensusTract_poly")
iwidth = 600
iheight = 400
```

6. Next, we need to determine the index of the population field so that we can get the population count for each area:

```
pop_index = None
dots = []
for i, f in enumerate(inShp.fields):
    if f[0] == "POPULAT11":
        # Account for deletion flag
        pop_index = i-1
```

7. Then, we calculate the population density value. We want to create a dot on the map for every 100 people:

```
for sr in inShp.shapeRecords():
    population = sr.record[pop_index]
    density = population / 100
    found = 0
```

8. We will loop through each polygon and randomly distribute the points to create a density map:

```
while found < int(density):
    minx, miny, maxx, maxy = sr.shape.bbox
    x = random.uniform(minx, maxx)
    y = random.uniform(miny, maxy)
    if point_in_poly(x, y, sr.shape.points):
        dots.append((x, y))
        found += 1
```

9. We're now ready to create our output image:

```
c.color = (255, 0, 0, 0xff)
for d in dots:
    x, y = world2screen(inShp.bbox, iwidth, iheight, *d)
    c.filled_rectangle(x-1, y-1, x+1, y+1)
```

10. Our dots have been created. Now, we need to create the outlines of the census tract:

```
c.color = (0, 0, 0, 0xff)
for s in inShp.iterShapes():
    pixels = []
    for p in s.points:
```

```
pixel = world2screen(inShp.bbox, iwidth, iheight, *p)
pixels.append(pixel)
c.polyline(pixels)
```

11. Finally, we'll save the output image:

```
with open("DotDensity.png", "wb") as img:
    img.write(c.dump())
```

This script outputs an outline of the census tract, along with the density dots, to show population concentration very effectively:

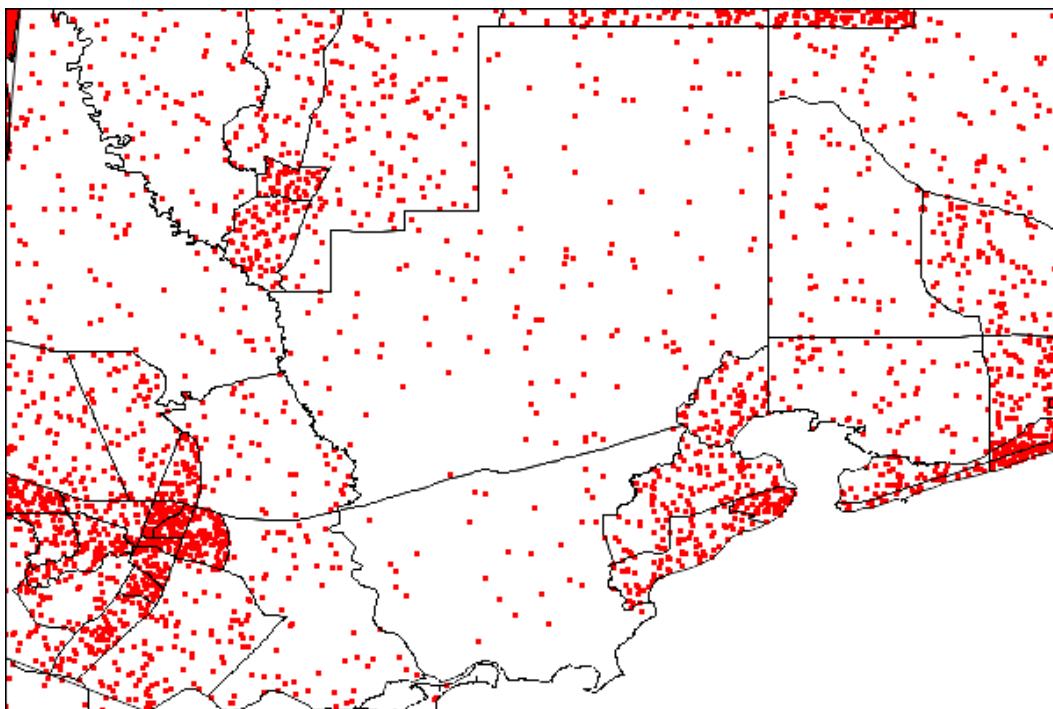


Figure 6.8 – A population dot density map created with pure Python

Now, let's check out the second type of map: choropleth maps.

## Choropleth maps

A choropleth uses shading, coloring, or symbols to show an average value or quantity within an area. It makes it easy for us to visualize large amounts of data as a summary. This method is useful if related data spans multiple polygons. For example, in a worldwide population density map by country, many countries have disconnected polygons (for example, Hawaii is an island state of the US).

In this example, we'll use the **Python Imaging Library (PIL)** we discussed in *Chapter 3, The Geospatial Technology Landscape*. PIL is not purely Python but is designed specifically for Python. We'll recreate our previous dot density example as a choropleth map. We'll calculate a density ratio for each census tract based on the number of people (population) per square kilometer and use that value to adjust the color. Dark is more densely populated while lighter is less. Follow these steps:

1. First, we will import our libraries:

```
import math
import shapefile
try:
    import Image
    import ImageDraw
except:
    from PIL import Image, ImageDraw
```

2. Then, we'll need our geospatial coordinates to image coordinates conversion function:

```
def world2screen(bbox, w, h, x, y):
    """convert geospatial coordinates to pixels"""
    minx, miny, maxx, maxy = bbox
    xdist = maxx - minx
    ydist = maxy - miny
    xratio = w/xdist
    yratio = h/ydist
    px = int(w - ((maxx - x) * xratio))
    py = int((maxy - y) * yratio)
    return (px, py)
```

3. Now, we open our shapefile and set our output image size:

```
inShp = shapefile.Reader("GIS_CensusTract_poly")
iwidth = 600
iheight = 400
```

4. We then set up PIL to draw our map image:

```
img = Image.new("RGB", (iwidth, iheight), (255, 255, 255))
draw = ImageDraw.Draw(img)
```

5. Just like our previous example, we need to get the index of the population field:

```
pop_index = None
area_index = None
for I, f in enumerate(inShp.fields):
    if f[0] == "POPULAT1":
```

```

pop_index = i-1
elif f[0] == "AREASQK":
    area_index = i-1

```

6. Now, we can draw the polygons, shade them according to population density, and save the image:

```

for sr in inShp.shapeRecords():
    density = sr.record[pop_index]/sr.record[area_index]
    weight = min(math.sqrt(density/80.0), 1.0) * 50
    R = int(205-- weight)
    G = int(215-- weight)
    B = int(245-- weight)
    pixels = []
    for x, y in sr.shape.points:
        (px, py) = world2screen(inShp.bbox, iwidth, iheight, x,
y)
        pixels.append((px, py))
    draw.polygon(pixels, outline=(255, 255, 255), fill=(R, G,
B))
Img.save("choropleth.png")

```

This script produces the following diagram with the relative density of tracts. You can adjust the color using the R, G, and B variables:

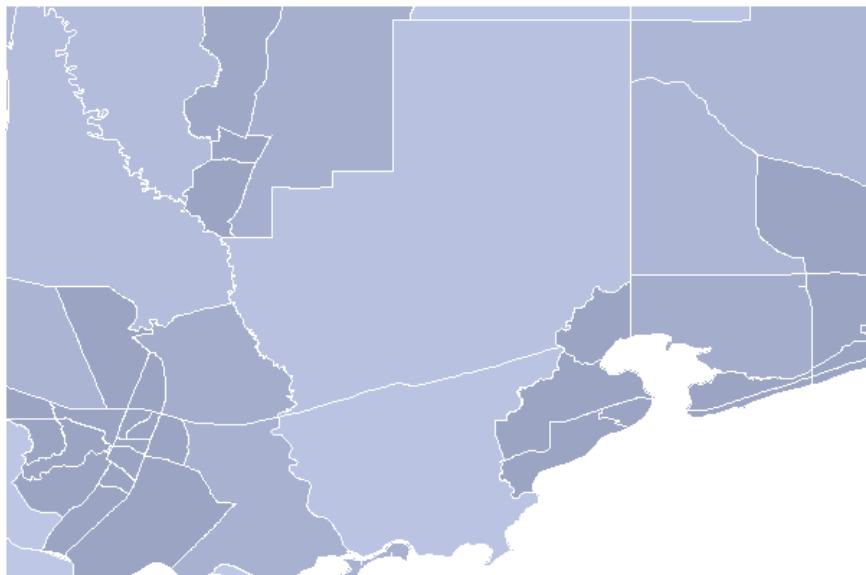


Figure 6.9 – Population density in a Python-generated choropleth map  
with darker areas representing higher population by area

Now that we can show statistical data from shapefiles, we can look at a statistical data source that is even more common than shapefiles: spreadsheets.

## Using spreadsheets

Spreadsheets such as Microsoft Office Excel and Open Office Calc are inexpensive (even free), ubiquitous, easy to use, and great for recording structured data. For these reasons, spreadsheets are widely used to collect data for entry into a GIS format. As an analyst, you will find yourself working with spreadsheets frequently.

In the previous chapters, we discussed the CSV format, which is a text file with the same basic rows and columns data structure as a spreadsheet. For CSV files, you use Python's built-in `csv` module. But most of the time, people don't bother exporting a true spreadsheet to a generic CSV file. That's where the pure Python `xlrd` module comes into play. The name `xlrd` is short for Excel Reader and is available from PyPI. There is also an accompanying module, `xlwt` (Excel Writer), for writing spreadsheets. These two modules make reading and writing Excel spreadsheets a snap. Combine it with `PyShp` and you can move back and forth between spreadsheets and shapefiles with ease. This example demonstrates converting a spreadsheet into a shapefile. We'll use a spreadsheet version of the New York City museum point data available at [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_06\\_Asset\\_Files/NYC\\_MUSEUMS\\_GEO.xls](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_06_Asset_Files/NYC_MUSEUMS_GEO.xls).

The spreadsheet contains the attribute data, followed by an `x` column with the longitude and a `y` column with the latitude. To export it to a shapefile, we'll execute the following steps:

1. Open the spreadsheet.
2. Create a shapefile `Writer` object.
3. Capture the first row of the spreadsheet as the `dbf` columns.
4. Loop through each row of the spreadsheet and copy the attributes to `dbf`.
5. Create a point from the `x` and `y` spreadsheet columns.

The script is as follows:

```
import xlrd
import shapefile
xls = xlrd.open_workbook("NYC_MUSEUMS_GEO.xls")
sheet = xls.sheet_by_index(0)
with shapefile.Writer("NYC_MUSEUMS_XLS2SHP",
shapefile.POINT) as w:
    for I in range(sheet.ncols):
        w.field(str(sheet.cell(0, i).value), "C", 40)
    for i in range(1, sheet.nrows):
        values = []
```

```

for j in range(sheet.ncols):
    values.append(sheet.cell(i, j).value)
w.record(*values)
w.point(float(values[-2]), float(values[-1]))

```

Converting a shapefile into a spreadsheet is a much less common operation, though not difficult. To convert a shapefile into a spreadsheet, you need to make sure you have an x and a y column by using the *Adding fields* example from the *Editing shapefiles* section in this chapter. You would loop through the shapes and add the x, y values to those columns.

Then, you would read the field names and column values from dbf into an xlwt spreadsheet object or a CSV file using the csv module. The coordinate columns are labeled in the following screenshot:

	A	B	C	D	E	F	G	H	I
1	NAME	TEL	URL	ADDRESS1	ADDRESS2	CITY	ZIP	XCOORD	YCOORD
2	Alexander	(212) 514-	http://ww	1 Bowling Grn		New York	10004	-74.0138	40.70382
3	Alice Aust	(718) 816-	http://			Staten Isla	10305	-74.063	40.61512
4	American	(212) 368-	http://			New York	10032	-73.9473	40.83385
5	American	(212) 265-	http://			New York	10019	-73.9781	40.76163
6	American	(212) 363-	http://ww	Ellis Island		New York	0	-74.0397	40.69906
7	American	(212) 769-	http://ww	Ce		New York	10024	-73.9736	40.78083
8	American	(212) 571-	http://ww	75		New York	10013	-74.007	40.72353
9	Americas	(212) 249-	http://as.e68			New York	10021	-73.966	40.76883
10	Anne Fran	(212) 431-	http://ww	38 Crosby	5th Floor	New York	10012	-73.9996	40.72113
11	Asia Socie	(212) 288-	http://ww	725 Park Avenue		New York	10021	-73.9643	40.76983
12	Audubon Terrace		http://ww	Broadway at 155 Stre	New York		10032	-73.9465	40.83358
13	Bartow-Pe	(718) 885-	http://bar	895 Shore	Pelham Ba	Bronx	10464	-73.8056	40.87178
14	Bowne Ho	(718) 359-	http://ww	37-01 Bowne Street		Queens	11354	-73.8249	40.76289
15	Bronx Hist	(718) 881-	http://ww	3309 Bainbridge Ave	Bronx		10467	-73.8797	40.87824
16	Bronx Mu	(718) 681-	http://ww	1040 Grand Concourse	Bronx		10456	-73.9198	40.83103

Figure 6.10 – We turned a spreadsheet into a shapefile but you can also go the other direction and turn a shapefile into a spreadsheet

In the next section, we'll use a spreadsheet as an input data source.

## Creating heat maps

A heat map is used to show the geographic clustering of data using a raster image that shows density. The clustering can also be weighed by using a field in the data to not only show geographic density but also an intensity factor. In this example, we'll use bear sighting data contained in the CSV dataset, which stores the data as points, to create a heat map of the frequency of bear sightings in different areas of Mississippi. This dataset is so simple that we're going to treat the CSV file as a text file, which is one of the nice features of a CSV file.

You can download the dataset from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_06\\_Asset\\_Files/bear\\_sightings.csv](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_06_Asset_Files/bear_sightings.csv).

The output is going to be a simple HTML web map that you can open in any web browser. The web map will be based on the excellent Leaflet JavaScript library. On top of that, we'll use the Folium Python library, which makes it easy for us to create Leaflet web maps, in order to generate the HTML page:

```
import os
import folium
from folium.plugins import HeatMap
f = open('bear_sightings.csv', 'r')
lines = f.readlines()
lines.pop(0)
data = []
bears = [list(map(float, l.strip().split(','))) for l in lines]
m = folium.Map([32.75, -89.52], tiles='stamentoner', zoom_start=7,
max_zoom=7, min_zoom=7)
HeatMap(bears, max_zoom=16, radius=22, min_opacity=1, blur=30).add_to(m)
m.save('heatmap.html')
```

This script will create a file called `heatmap.html`. Open it in any web browser to see an image similar to the following figure:

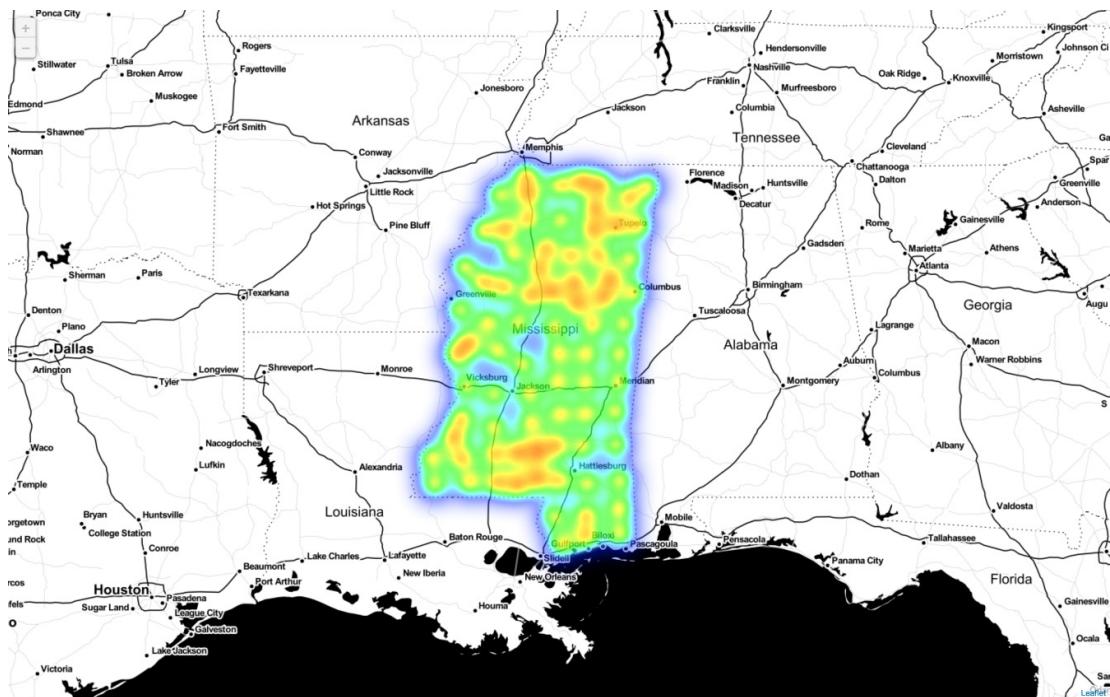


Figure 6.11 – A heat map generated with the folium library is an excellent way to create a distribution density map, as in this example of bear sightings

Next, we'll learn how to use data generated by a GPS to collect field data such as the information in the preceding heat map.

## Using GPS data

The most common type of GPS data these days is the Garmin GPX format. We covered this XML format, which has become an unofficial industry standard, in *Chapter 4, Geospatial Python Toolbox*. Because it is an XML format, all of the well-documented rules of XML apply to it. However, there is another type of GPS data that pre-dates XML and GPX, called **National Marine Electronics Association (NMEA)** data. This data is ASCII text sentences that are designed to be streamed.

You occasionally bump into this format from time to time because even though it is older and esoteric, it is still very much alive and well, especially for communicating ship locations via the **Automated Identification System (AIS)**, which tracks ships globally. But as usual, you have a good option in pure Python. The `pynmea` module is available on PyPI. The following code is a small sample of NMEA sentences:

```
$GPRMC,012417.859,V,1856.599,N,15145.602,W,12.0,7.27,020713,,E\*4F
$GPGGA,012418.859,1856.599,N,15145.602,W,0,00,,,M,,M,,\*54
```

```
$GPGLL,1856.599,N,15145.602,W,012419.859,V\*35  
$GPVTG,7.27,T,,M,12.0,N,22.3,K\*52  
$GPRMC,012421.859,V,6337.596,N,12330.817,W,66.2,23.41,020713,,E\*74
```

To install the `pynmea` module from PyPI and download the complete sample file, you can visit the following URL: [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_06\\_Asset\\_Files/nmea.txt](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_06_Asset_Files/nmea.txt).

Then, you can run the following code, which will parse the NMEA sentences into objects. The NMEA sentences contain a wealth of information:

```
from pynmea.streamer import NMEAStream  
nmeaFile = open("nmea.txt")  
nmea_stream = NMEAStream(stream_obj=nmeaFile)  
next_data = nmea_stream.get_objects()  
nmea_objects = []  
while next_data:  
    nmea_objects += next_data  
    next_data = nmea_stream.get_objects()  
for nmea_ob in nmea_objects:  
    if hasattr(nmea_ob, "lat"):  
        print("Lat/Lon: ({}, {})".format(nmea_ob.lat, \  
                                         nmea_ob.lon))
```

The latitudes and longitudes are stored in a format called **degrees decimal minutes**. For example, this random coordinate, 4533.35, is 45 degrees and 33.35 minutes. 0.35 of a minute is exactly 21 seconds. In another example, 16708.033 is 167 degrees and 8.033 minutes. 0.033 of a minute is approximately 2 seconds. You can find more information about the NMEA format at <http://aprs.gids.nl/nmea/>.

GPS data is an important location data source, but there's another way we can describe a point on the Earth using a street address. The method for locating a street address on the Earth is called geocoding.

## Turning addresses into points with geocoding

Geocoding is the process of converting a street address into latitude and longitude. This operation is critical to in-vehicle navigation systems and online driving direction websites. Python has two excellent geocoder libraries available named `geocoder` and `geopy`. Both take advantage of online geocoding services to allow you to geocode addresses programmatically. The `geopy` library even lets you reverse geocode to match a latitude and longitude to the nearest address.

First, let's do a quick example with the geocoder library, which defaults to using **OpenStreetMaps** as its engine:

```
import geocoder
g = geocoder.osm("1403 Washington Ave, New Orleans, LA 70130")
print(g.geojson)
print()
print(g.wkt)
```

This will give us the following:

```
{"type": "Feature", "geometry": {"type": "Point", "coordinates": [-90.0842184999999, 29.9287839]},  
"bbox": {"northeast": [29.9301328802915, -90.0828695197085],  
"southwest": [29.9274349197085, -90.0855674802915]},  
"properties": {"quality": "street_address", "lat": 29.9287839, "city": "New Orleans",  
"provider": "google", "geometry": {"type": "Point", "coordinates": [-90.0842184999999, 29.9287839]},  
"lng": -90.0842184999999, "method": "geocode", "encoding": "utf-8",  
"confidence": 9, "address": "1403 Washington Ave,  
New Orleans, LA 70130, USA", "ok": True, "neighborhood": "Garden  
District", "county": "Orleans Parish",  
"accuracy": "ROOFTOP", "street": "Washington Ave", "location": "1403  
Washington Ave, New Orleans, LA 70130",  
"bbox": {"northeast": [29.9301328802915, -90.0828695197085],  
"southwest": [29.9274349197085, -90.0855674802915]},  
"status": "OK", "country": "US", "state": "LA", "housenumber": "1403",  
"postal": "70130"}}
```

Then we can print just the point as a WKT geometry:

```
print(g.wkt)
```

To return the point:

```
'POINT (-90.0842184999999 29.9287839)'
```

Here, we print the GeoJSON record for that address that contains all known information in the OpenStreetMap database. Then, we print out the returned latitude and longitude as a WKT string, which could be used as input to other operations such as checking whether the address is inside of a floodplain polygon. The documentation for this library also shows you how to switch to other online geocoding services such as Bing or Yahoo. Some of these services require an API key and may have request limits.

Now, let's look at the geopy library. In this example, we'll geocode against the OpenStreetMap database again. Once we match the address to a location, we'll turn around and reverse geocode it. Note that we have to specify a user agent. This is the name of our application, which we'll just call **LGAWP**, short for *Learning Geospatial Analysis with Python*:

```
from geopy.geocoders import Nominatim
g = Nominatim(user_agent="LGAWP")
location = g.geocode("88360 Diamondhead Dr E, Diamondhead, MS 39525")
print(location.raw)
```

This will return the following GeoJSON string:

```
{'lon': '-89.3462139', 'osm_type': 'node',
'type': 'yes', 'place_id': '25470846',
'display_name': 'NVision Solutions Inc.,
88360, Diamondhead Drive East, Diamondhead,
Hancock County, Mississippi, 39520,
United States of America', 'boundingbox':
['30.3961462', '30.3962462', '-89.3462639',
'-89.3461639'], 'licence': 'Data ©
OpenStreetMap contributors, ODbL 1.0.
http://www.openstreetmap.org/copyright', 'osm_id':
'2470309304', 'importance': 0.421,
'class': 'office', 'lat': '30.3961962'}
```

Now that we know of several different ways to geocode, let's look at speeding up the process. If you have thousands of addresses to geocode, it can take a while. Using multiprocessing, you can shorten a geocoding process that might take days into a few hours.

## Performing GIS analysis faster with multiprocessing

Geospatial datasets are very large. Processing them can take time, which can be hours or sometimes even days. But there's a way you can speed processing up for certain operations. Python's built-in multiprocessing module can spawn multiple processes on your computer to take advantage of all of the available processors.

One operation that works really well with the multiprocessing module is geocoding. In this example, we'll geocode a list of cities and split that processing across all of the processors on your machine. We'll use the same geocoding technique as before, but this time, we'll add the multiprocessing module to increase the potential for greater speed and scalability. The following code will geocode a list of cities simultaneously across multiple processors:

1. First, we import the modules we need:

```
from geopy.geocoders import Nominatim
```

2. Next, we create our geocoder object, giving it the name of our application:

```
g = Nominatim(user_agent="LGAWP")
```

3. Now, we need a function to geocode an individual address:

```
def gcode(address):  
    location = g.geocode(address)  
    print("Geocoding: {}".format(address))  
    return location
```

4. Now, we have to create a main function to safely use the multiprocessing module:

```
if __name__ == '__main__':  
    import multiprocessing as mp
```

5. Next, we create our list of cities to process:

```
cities = ["New Orleans, LA", "Biloxi, MS", "Memphis, TN",  
"Atlanta, GA", "Little Rock, AR", "Destin, FL"]
```

6. Then, we set up our processor pool based on the number of processors available:

```
pool = mp.Pool(processes=mp.cpu_count())
```

7. Next, we map our list of cities to the geocode function through the processor pool:

```
results = pool.map(gcode, cities)
```

8. Then, we can print the results:

```
print(results)
```

This will give us the location of each city:

```
[Location(New Orleans, Orleans Parish, Louisiana, USA,  
(29.9499323, -90.0701156, 0.0)),  
Location(Biloxi, Harrison County, Mississippi, USA, (30.374673,  
-88.8459433348286, 0.0)),  
Location(Memphis, Shelby County, Tennessee, USA, (35.1490215,  
-90.0516285, 0.0)),  
Location(Atlanta, Fulton County, Georgia, USA, (33.7490987,  
-84.3901849, 0.0)),  
Location(Little Rock, Arkansas, USA, (34.7464809, -92.2895948,  
0.0)),  
Location(Destin, Okaloosa County, Florida, USA, (30.3935337,  
-86.4957834, 0.0))]
```

This technique can be very powerful, but not every type of processing can be performed this way. The type of processing you use has to support operations that can be broken apart into discrete calculations. But when you can break problems apart, like we did in this example, the results are orders of magnitude faster.

## Summary

In this chapter, we learned how to edit shapefiles and perform spatial and attribute selections. We created thematic maps from scratch using only Python. We also imported data from spreadsheets. Then, we parsed GPS data from NMEA streams. We used geocoding to convert street addresses into locations and back. And finally, we learned how to speed things up with parallel processing.

As a geospatial analyst, you may be familiar with both GIS and remote sensing, but most analysts specialize in one field or the other. That is why this book approaches the fields in separate chapters – so that we can focus on their differences. As we mentioned in the introduction, the techniques in this chapter are the building blocks for all geospatial analysis and will give you the tools you need so that you can learn about any aspect of this field.

In the next chapter, we'll dive into remote sensing. In GIS, we have been able to explore this field using pure Python modules. In remote sensing, we'll become more dependent on bindings to compiled modules written in C due to the sheer size and complexity of the data.

## Questions

1. What's the operation called for combining multiple features into a single feature?
2. What are the three types of thematic maps for showing population distribution and density?
3. What is the term for turning a street address into geospatial coordinates?

## Further reading

- <https://github.com/gisalgs/geom>
- <https://realpython.com/python-folium-web-maps-from-data/>



# 7

## Python and Remote Sensing

In this chapter, we will discuss remote sensing. Remote sensing is about gathering a collection of information about the Earth without making physical contact with it. Typically, this means having to use satellite or aerial imagery, **Light Detection and Ranging (LiDAR)**, which measures laser pulses from an aircraft to the Earth, or synthetic aperture radar. Remote sensing can also refer to computer processing of data that's been collected, which is how we'll use the term in this chapter. Remote sensing grows in exciting new ways every day as more satellites are launched and the distribution of data becomes easier. The high availability of satellite and aerial images, as well as interesting new types of sensors launching each year, is changing the role that remote sensing plays in understanding our world.

In remote sensing, we step through each pixel in an image and perform some form of query or mathematical process. An image can be thought of as a large numerical array. In remote sensing, these arrays can be quite large, in the order of tens of megabytes to several gigabytes in size. While Python is fast, only C-based libraries can provide the speed that's needed to loop through arrays at a tolerable speed.

We'll use the **Python Imaging Library (PIL)** for image processing and NumPy, which provides multidimensional array mathematics. While written in C for speed, these libraries are designed for Python and provide a Pythonic API.

In this chapter, we'll cover the following topics:

- Examining raster data properties
- Swapping image bands
- Creating image histograms
- Clipping images
- Classifying images
- Extracting features from images
- Understanding change detection
- Extracting image footprints using ChatGPT

First, we'll start with basic image manipulation and then build on each exercise, all the way to automatic change detection. These techniques will complement the previous chapters by adding the ability to process satellite data and other remote-sensing products to our toolbox.

## Technical requirements

The code files for this chapter are available on GitHub: [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730\\_07\\_Asset\\_Files](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730_07_Asset_Files).

## Examining raster data properties

As a geospatial analyst, understanding the metadata and properties of these raster images is crucial as they provide insights into the image's spatial reference, resolution, number of bands, data type, and other essential attributes. This information is vital for ensuring that the raster data is compatible with other datasets, aligns correctly within a spatial analysis, and is suitable for the intended analytical methods.

The following script serves as a practical tool for examining the metadata and properties of a raster image using the GDAL library in Python. By running this code, you can quickly assess the characteristics of a raster file, such as its projection, size, band properties, and more. This information is not only valuable for initial data exploration but also plays a critical role in preprocessing and quality control. Whether you're integrating raster data with other spatial datasets, preparing it for analysis, or troubleshooting issues, this code offers a streamlined way to access the essential details of the raster data.

We'll use a satellite image that we'll also use for other examples called `FalseColor.tif`. You can download this image as a ZIP file from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_07\\_Asset\\_Files/FalseColor.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_07_Asset_Files/FalseColor.zip).

Let's step through looking at the raster properties:

1. First, import the GDAL library from the `osgeo` package. This allows us to work with raster data:

```
from osgeo import gdal
```

2. Next, enable exceptions in GDAL. This helps us handle any errors that might occur during the execution of our code:

```
gdal.UseExceptions()
```

3. Then, define the path to our raster file, which in this case is named `FalseColor.tif`:

```
raster_file = 'FalseColor.tif'
```

4. After that, open the raster file using the `Open` method in GDAL, which returns a dataset object:

```
dataset = gdal.Open(raster_file)
```

5. Next, retrieve and print the metadata of the raster file. This includes information such as the file's author, creation date, and more:

```
metadata = dataset.GetMetadata()  
for key, value in metadata.items():  
    print(f'{key}: {value}')
```

6. Then, print the raster's projection, number of bands, and size. These properties describe the spatial reference, the number of layers, and the dimensions of the raster:

```
print(f'Projection: {dataset.GetProjection()}')  
print(f'Number of bands: {dataset.RasterCount}')  
print(f'Size: {dataset.RasterXSize} x {dataset.RasterYSize}')
```

7. Finally, loop through each band in the raster and print its properties. This includes the band's data type, minimum and maximum values, NoData value, and color table:

```
for band_num in range(1, dataset.RasterCount + 1):  
    band = dataset.GetRasterBand(band_num)  
    print(f'Band {band_num} Type: '\n         f'{gdal.GetDataTypeName(band.DataType)}')  
    print(f'Min: {band.GetMinimum()}, Max: '\n         f'{band.GetMaximum()}')  
    print(f'NoData Value: {band.GetNoDataValue()}')  
    print(f'Color Table: {band.GetColorTable()}')
```

When you run this code, you'll get the following output describing the different aspects of this satellite image in table format:

AREA_OR_POINT	Area
Projection	GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID["WGS 84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],AUTHORITY ["EPSG","6326"]],PRIMEM["Greenwich",0,AUTHORITY["EPSG", "8901"]],UNIT["degree",0.0174532925199433,AUTHORITY["EPSG", "9122"]],AXIS["Latitude",NORTH],AXIS["Longitude",EAST], AUTHORITY["EPSG", "4326"]]
Number of bands	3
Size	2,592 x 2,693
Band 1 Type	Byte
Min: 5.0, Max	209.0
NoData Value	None
Color Table	None
Band 2 Type	Byte
Min: 1.0, Max	179.0
NoData Value	None
Color Table	None
Band 3 Type	Byte
Min: 48.0, Max	255.0
NoData Value	None
Color Table	None

Table 7.1 – The output from viewing raster properties with GDAL

Now that we've explored the metadata and properties of a raster image, we have a solid understanding of its structure and content. Next, we'll dive into a more advanced operation that's often required in remote sensing analysis: swapping image bands. This technique allows us to rearrange the order of the bands within a multi-band image, enabling customized visualizations and analyses.

## Swapping image bands

Our eyes can only see colors in the visible spectrum as combinations of **red, green, and blue (RGB)**. Air- and space-borne sensors can collect wavelengths of the energy outside of the visible spectrum. To view this data, we must move images representing different wavelengths of light reflectance in and out of the RGB channels to make color images.

These images often end up as bizarre and alien color combinations that can make visual analysis difficult. An example of a typical satellite image is shown in the following Landsat 7 satellite scene along the Gulf of Mexico, which is a leading center for remote sensing and geospatial analysis in general:

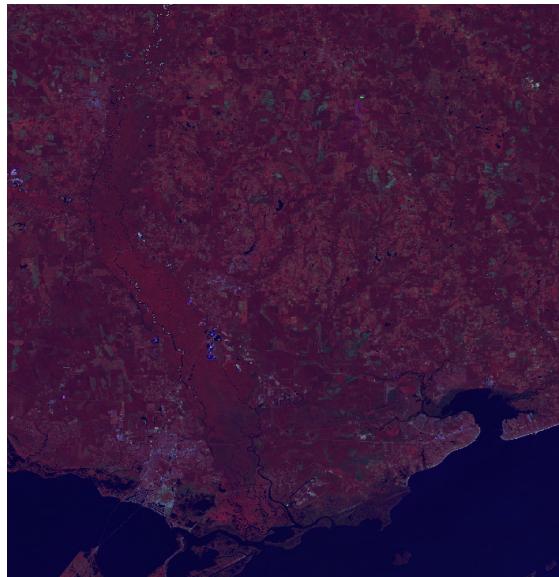


Figure 7.1 – A false-color satellite image along the Gulf of Mexico

Most of the vegetation appears red and the water appears almost black. This image is a type of false-color image, meaning the color of the image is not based on the RGB light. However, we can change the order of the bands or swap out certain bands to create another type of false-color image that looks more like the world we are used to seeing. To do so, you first need to download this image as a ZIP file from [https://github.com/PacktPublishing/Learning-Geospatial-Analyses-with-Python-Fourth-Edition/raw/main/B19730\\_07\\_Asset\\_Files/FalseColor.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analyses-with-Python-Fourth-Edition/raw/main/B19730_07_Asset_Files/FalseColor.zip). The GDAL library includes a module called `gdal_array` that loads and saves remotely-sensed images to and from NumPy arrays for easy manipulation. GDAL is a data access library and does not provide much in the name of processing. So, in this chapter, we will rely heavily on NumPy to change images for purposes such as swapping bands, clipping them, performing change detection, and more.

In this example, we'll load the image into a NumPy array using `gdal_array` and then we'll immediately save it back to a new GeoTiff file. However, upon saving, we'll use NumPy's advanced array-slicing feature to change the order of the bands. Images in NumPy are multi-dimensional arrays in the order of band, height, and width. This means that an image with three bands will be an array of length 3, containing an array for the band, height, and width of the image. It's important to note that NumPy references array locations as  $y, x$  (row, column) instead of the usual  $x, y$  (column, row) format we work with in spreadsheets and other software. Let's get started:

1. First, import `gdal_array`:

```
from osgeo import gdal_array
```

2. Next, load an image named `FalseColor.tif` into a NumPy array:

```
# name of our source image
src = "FalseColor.tif"
arr = gdal_array.LoadFile(src)
```

3. Next, reorder the image bands by slicing the array, rearranging the order, and saving it back out:

```
output = gdal_array.SaveArray(arr[[1, 0, 2], :], \
"swap.tif", format="GTiff", prototype=src)
output = None
```

In the `SaveArray()` method, the last argument is called a prototype. This argument lets you specify another image for GDAL from which you copy spatial reference information and some other image parameters. Without this argument, we'd end up with an image without georeferencing information, which could not be used in a GIS. In this case, we specify our input image filename because the images are identical, except for the band order. In this method, you can tell that the Python GDAL API is a wrapper around a C library and is not as Pythonic as a Python-designed library. For example, a pure Python library would have written the `SaveArray()` method as `save_array()` to follow Python standards.

The result of this example produces the `swap.tif` image, which is a much more visually appealing image with green vegetation and blue water:



Figure 7.2 – A satellite image with the band order altered

There's only one problem with this image: it's kind of dark and difficult to see. Let's see if we can figure out why in the next section.

## Creating image histograms

A histogram shows the statistical frequency of data distribution within a dataset. In the case of remote sensing, the dataset is an image. The data distribution is the frequency of pixels in the range of 0 to 255, which is the range of 8-byte numbers that are used to store image information on computers.

In an RGB image, color is represented as a three-digit tuple, with `(0, 0, 0)` being black and `(255, 255, 255)` being white. We can graph the histogram of an image with the frequency of each value along the  $x$  axis and the range of 256 possible pixel values along the  $x$  axis.

Remember in *Chapter 1, Learning about Geospatial Analysis with Python*, in the *Creating the simplest possible Python GIS* section, when we used the Turtle graphics engine included with Python to create a simple GIS? Well, we can also use it to easily graph histograms.

Histograms are usually a one-off product that makes a quick script. Also, histograms are typically displayed as a bar graph with the width of the bars representing the size of grouped data bins. But, in an image, each bin has only one value, so we'll create a line graph. We'll use the `histogram` function in this example and create an RGB line for each respective band.

The graphing portion of this example also defaults to scaling the  $y$  axis values to the max RGB frequency found in the image. Technically, the  $y$  axis represents the maximum frequency, which is the number of pixels in the image, which would be the case if the image was all one color. We'll use the `turtle` module again here, but this example could be easily converted into any graphical output module. Let's take a look at the `swap.tif` image we created in the previous example:

1. First, let's import the libraries we need, including the `turtle` graphics library:

```
from osgeo import gdal_array
import turtle as t
```

2. Now, create a `histogram` function that can take an array and sort the numbers into bins making up the histogram:

```
def histogram(a, bins=list(range(0, 256))):
    fa = a.flat
    n = gdal_array.numpy.searchsorted(\n        gdal_array.numpy.sort(fa), bins)
    n = gdal_array.numpy.concatenate([n, [len(fa)]])
    hist = n[1:]-n[:-1]
    return hist
```

3. Finally, we have our `turtle` graphics function, which takes a histogram and draws it:

```
def draw_histogram(hist, scale=True):
```

4. Draw the graph axes using the following code:

```
t.color("black")
axes = ((-355, -200), (355, -200), \
        (-355, -200), (-355, 250))
t.up()
for p in axes:
    t.goto(p)
    t.down()
```

5. Then, we can label them:

```
t.goto(0, -250)
t.write("VALUE", font=("Arial, ", 12, "bold")) t.up()
t.goto(-400, 280)
t.write("FREQUENCY", font=("Arial, ", 12, "bold")) x = -355
y = -200
t.up()
```

6. Now, add tick marks to the  $x$  axis so that we can see the line values:

```
for i in range(1, 11): x = x+65
t.goto(x, y)
t.down() t.goto(x, y-10) t.up() t.goto(x, y-25)
t.write("{}".format((i*25)), align="center")
```

7. Let's do the same for the  $y$  axis:

```
x = -355
y = -200
t.up()
pixels = sum(hist[0]) if scale:
max = 0
for h in hist: hmax = h.max() if hmax > max:
max = hmax pixels = max
label = int(pixels/10) for i in range(1, 11):
y = y+45 t.goto(x, y)
t.down() t.goto(x-10, y) t.up()
t.goto(x-15, y-6) t.write("{}".format((i*label)), align="right")
```

8. Now, we can begin plotting our histogram lines:

```
x_ratio = 709.0 / 256 y_ratio = 450.0 / pixels
colors = ["red", "green", "blue"] for j in range(len(hist)):
    h = hist[j] x = -354
    y = -199
    t.up()
    t.goto(x, y)
    t.down() t.color(colors[j]) for i in range(256):
        x = i * x_ratio
        y = h[i] * y_ratio x = x - (709/2)
        y = y + -199
        t.goto((x, y))
```

9. Finally, we can load our image and plot its histogram using the functions we defined previously:

```
im = "swap.tif" histograms = []
arr = gdal_array.LoadFile(im) for b in arr:
    histograms.append(histogram(b)) draw_histogram(histograms)
t.pen(shown=False)
t.done()
```

Here's what the histogram for `swap.tif` looks like after running the preceding code example:

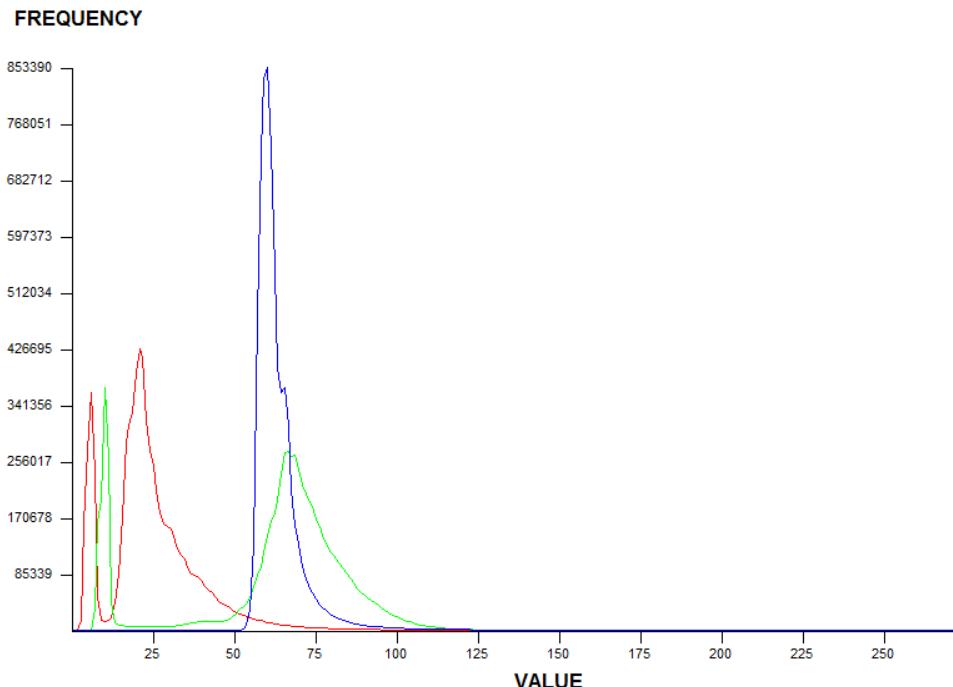


Figure 7.3 – A Python-generated satellite image histogram

As you can see, all three bands are grouped closely toward the left-hand side of the graph and all have values less than 125 or so. As these values approach zero, the image becomes darker, which is not surprising.

Just for fun, let's run the script again. When we call the `draw_histogram()` function, we'll add the `scale=False` option to get a sense of the size of the image and provide an absolute scale. We'll change the following line:

```
draw_histogram(histograms)
```

We'll change this to the following:

```
draw_histogram(histograms, scale=False)
```

This change will produce the following histogram graph:

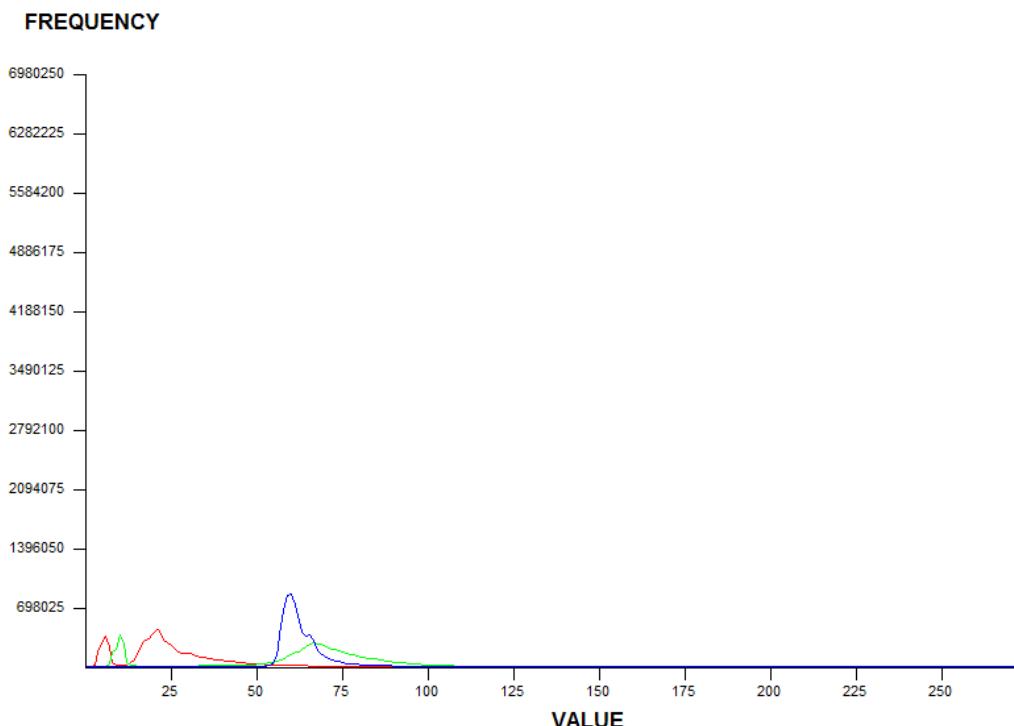


Figure 7.4 – An unscaled version of the histogram

As you can see, it's harder to see the details of the value distribution. However, this absolute-scale approach is useful if you are comparing multiple histograms of different products that were produced from the same source image.

So, now that we understand the basics of looking at an image statistically using histograms, how do we make our image brighter? Let's try a histogram stretch.

## Performing a histogram stretch

A histogram stretch operation does exactly what its name says – it redistributes the pixel values across the whole scale. By doing so, we have more values at the higher-intensity level and the image becomes brighter. So, in this example, we'll reuse our histogram function, but we'll add another function called `stretch()` that takes an image array, creates the histogram, and then spreads out the range of values for each band. We'll run these functions on `swap.tif` and save the result in an image called `stretched.tif`:

```
from osgeo import gdal_array
import operator
from functools import reduce

def histogram(a, bins=list(range(0, 256))):
    fa = a.flat
    n = gdal_array.numpy.searchsorted(\n        gdal_array.numpy.sort(fa), bins)
    n = gdal_array.numpy.concatenate([n, [len(fa)]])
    hist = n[1:] - n[:-1]
    return hist

def stretch(a):
    hist = histogram(a)
    lut = []
    for b in range(0, len(hist), 256):
        step = reduce(operator.add, hist[b:b+256]) / 255
        n = 0
        for i in range(256):
            lut.append(n / step)
            n = n + hist[i+b]
    gdal_array.numpy.take(lut, a, out=a)
    return a

src = "swap.tif"
arr = gdal_array.LoadFile(src)
stretched = stretch(arr)
output = gdal_array.SaveArray(arr, "stretched.tif", \
format="GTiff", prototype=src)
output = None
```

The stretch algorithm will produce the following figure. Look how much brighter and visually appealing it is:

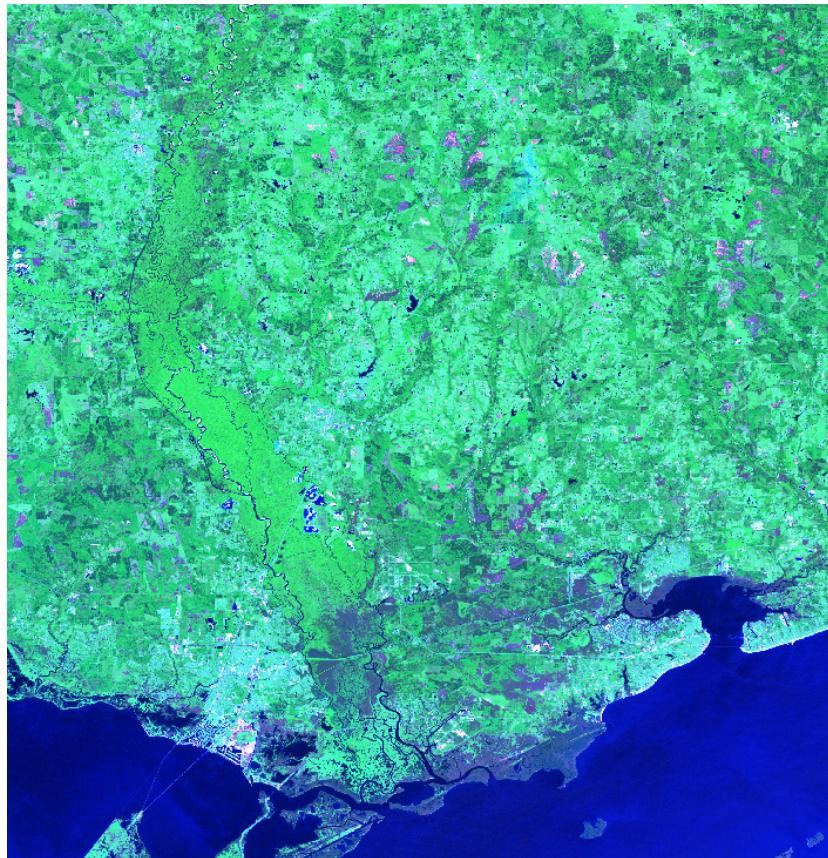


Figure 7.5 – The stretched version of the satellite image is much brighter

We can run our `turtle` graphics histogram script on `stretched.tif` by changing the filename in the `im` variable to `stretched.tif`:

```
im = "stretched.tif"
```

Running the preceding code will give us the following histogram:

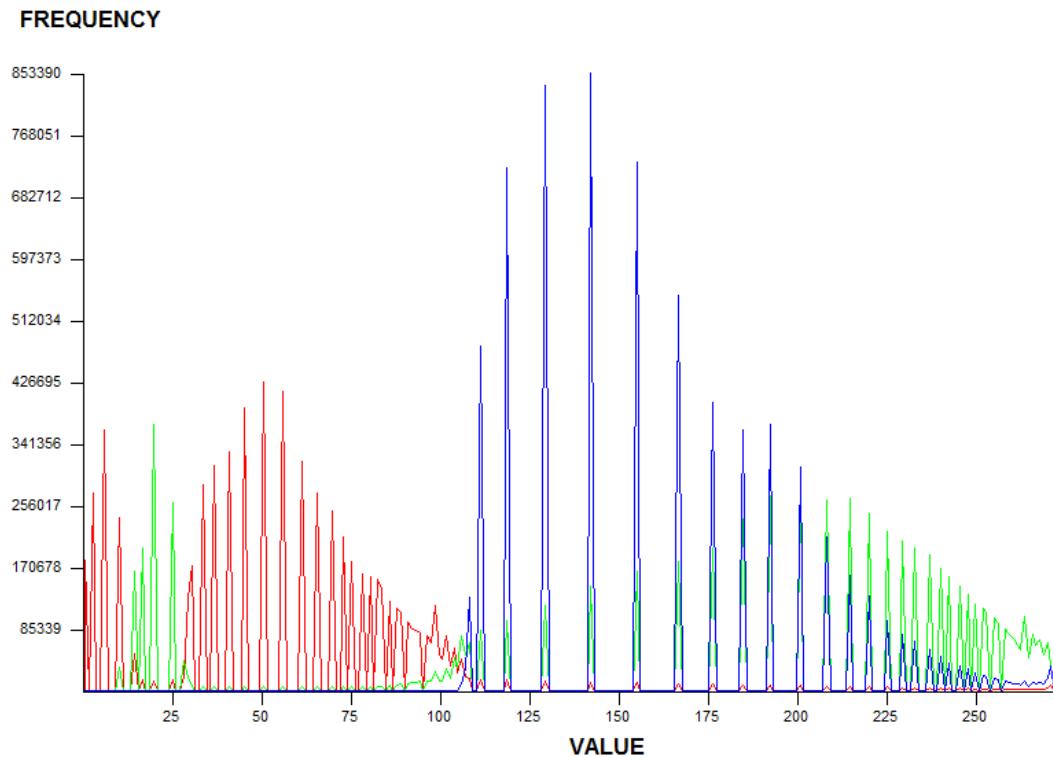


Figure 7.6 – The stretched histogram spreads the values both vertically and horizontally

As you can see, all three bands are distributed evenly now. Their relative distribution to each other is the same, but, within the image, they are now spread across the spectrum.

Now that we can change images for better presentation, let's look at clipping them to examine a particular area of interest.

## Clipping images

Very rarely is an analyst interested in an entire satellite scene, which can easily cover hundreds of square miles. Given the size of satellite data, we are highly motivated to reduce the size of an image to only our area of interest. The best way to accomplish this reduction is to clip an image to a boundary that defines our study area. We can use shapefiles (or other vector data) as our boundary definition and get rid of all the data outside that boundary.

The following figure contains our `stretched.tif` image with a county boundary file layered on top, visualized in **Quantum GIS (QGIS)**:

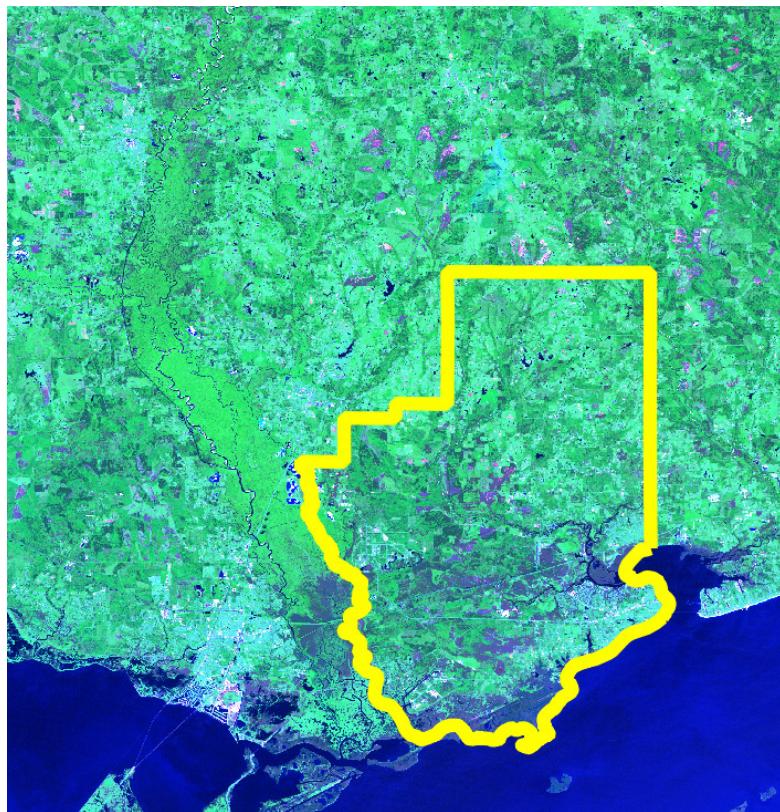


Figure 7.7 – Our satellite image with a county boundary shapefile overlayed

To clip the image, we need to follow these steps:

1. Load the image into an array using `gdal_array`.
2. Create a shapefile reader using `PyShp`.
3. Rasterize the shapefile into a georeferenced image (convert it from a vector into a raster).
4. Turn the shapefile image into a binary mask or filter to only grab the image pixels we want within the shapefile boundary.
5. Filter the satellite image through the mask.
6. Discard satellite image data outside the mask.
7. Save the clipped satellite image as `clip.tif`.

We installed PyShp in *Chapter 4, Geospatial Python Toolbox*, so you should already have it installed from PyPi. We will also add a couple of useful new utility functions in this script. The first is `world2pixel()`, which uses the `GDAL GeoTransform` object to do the world-coordinate to image-coordinate conversion for us. GDAL is using the same process we've used throughout the rest of this book, but the image metadata is contained in the GDAL raster object.

We also add the `imageToArray()` function, which converts a PIL image into a NumPy array. The county boundary shapefile is the `hancock.shp` boundary we've used in previous chapters, but you can also download it here if you need to: [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_07\\_Asset\\_Files/hancock.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_07_Asset_Files/hancock.zip).

We're using PIL because it is the easiest way to rasterize our shapefile as a mask image to filter out the pixels beyond the shapefile boundary. Let's get started:

1. First, let's load the libraries we need:

```
import operator
from osgeo import gdal
from osgeo import gdal_array, osr
import shapefile
```

2. Now, let's load PIL. This may need to be installed slightly differently on different platforms, so we have to check for that difference:

```
try:
    import Image
    import ImageDraw
except:
    from PIL import Image, ImageDraw
```

3. Now, let's set up the variables for our input image, shapefile, and our output image:

```
raster = "stretched.tif"
shp = "hancock"
output = "clip"
```

4. Next, create a function that simply converts an image into a Numpy array so that we can convert the mask image we will create and use it in our NumPy-based clipping process:

```
def imageToArray(i):
    a = gdal_array.numpy.frombuffer(i.tobytes(), 'b')
    a.shape = i.im.size[1], i.im.size[0]
    return a
```

5. Next, we need a function to convert geospatial coordinates into image pixels, which will allow us to use coordinates from our clipping shapefile to limit which image pixels are saved:

```
def world2Pixel(geoMatrix, x, y):  
    ulX = geoMatrix[0]  
    ulY = geoMatrix[3]  
    xDist = geoMatrix[1]  
    yDist = geoMatrix[5]  
    rtnX = geoMatrix[2]  
    rtnY = geoMatrix[4]  
    pixel = int((x - ulX) / xDist)  
    line = int((ulY - y) / abs(yDist))  
    return (pixel, line)
```

6. Now, we can load our source image into a NumPy array:

```
srcArray = gdal_array.LoadFile(raster)
```

7. We'll also load the source image as a gdal image because gdal\_array does not give us the geotransform information we need to convert coordinates into pixels:

```
srcImage = gdal.Open(raster)  
geoTrans = srcImage.GetGeoTransform()
```

8. Now, use the Python `shapefile` library to open our shapefile:

```
r = shapefile.Reader("{}{}.shp".format(shp))
```

9. Next, convert the shapefile bounding box coordinates into image coordinates based on our source image:

```
minX, minY, maxX, maxY = r.bbox  
ulX, ulY = world2Pixel(geoTrans, minX, maxY)  
lrX, lrY = world2Pixel(geoTrans, maxX, minY)
```

10. Then, we can calculate the size of our output image based on the extents of the shapefile and take just that part of the source image:

```
pxWidth = int(lrX - ulX)  
pxHeight = int(lrY - ulY)  
clip = srcArray[:, ulY:lrY, ulX:lrX]
```

11. Next, create new geomatrix data for the output image:

```
geoTrans = list(geoTrans)  
geoTrans[0] = minX  
geoTrans[3] = maxY
```

12. Now, we can create a simple black-and-white mask image from the shapefile that will define the pixels we want to extract from the source image:

```
pixels = []
for p in r.shape(0).points:
    pixels.append(world2Pixel(geoTrans, p[0], p[1]))
rasterPoly = Image.new("L", (pxWidth, pxHeight), 1)
rasterize = ImageDraw.Draw(rasterPoly)
rasterize.polygon(pixels, 0)
```

13. Next, convert the mask image into a NumPy array:

```
mask = imageToArray(rasterPoly)
```

14. Finally, we're ready to use the mask array to clip the source array in NumPy and save it to a new geotiff image:

```
clip = gdal_array.numpy.choose(mask, \
(clip, 0)).astype(gdal_array.numpy.uint8)\noutput = gdal_array.SaveArray(clip,\n"{}.tif".format(output), format="GTiff", \
prototype=raster)\noutput = None
```

This script produces the following clipped image:

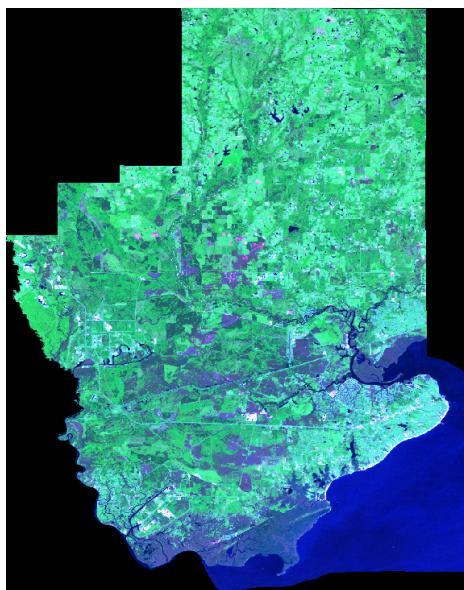


Figure 7.8 – The clipped satellite image

The areas that remain outside the county boundary that appear in black are called NoData values, meaning there is no information at that location, and they are ignored by most geospatial software. Because images are rectangular, the NoData values are common for data that does not completely fill an image.

You have now walked through an entire workflow that is used by geospatial analysts around the world every day to prepare multispectral satellite and aerial images for use in a GIS. We'll look at how we can analyze images as information in the next section.

## Classifying images

**Automated remote sensing (ARS)** is rarely ever done in the visible spectrum. ARS processes images without any human input. The most commonly available wavelengths outside of the visible spectrum are infrared and near-infrared.

The following figure shows a thermal image (band 10) from a fairly recent Landsat 8 flyover of the US Gulf Coast from New Orleans, Louisiana, to Mobile, Alabama. The major natural features in the image have been labeled so that you can orient yourself:

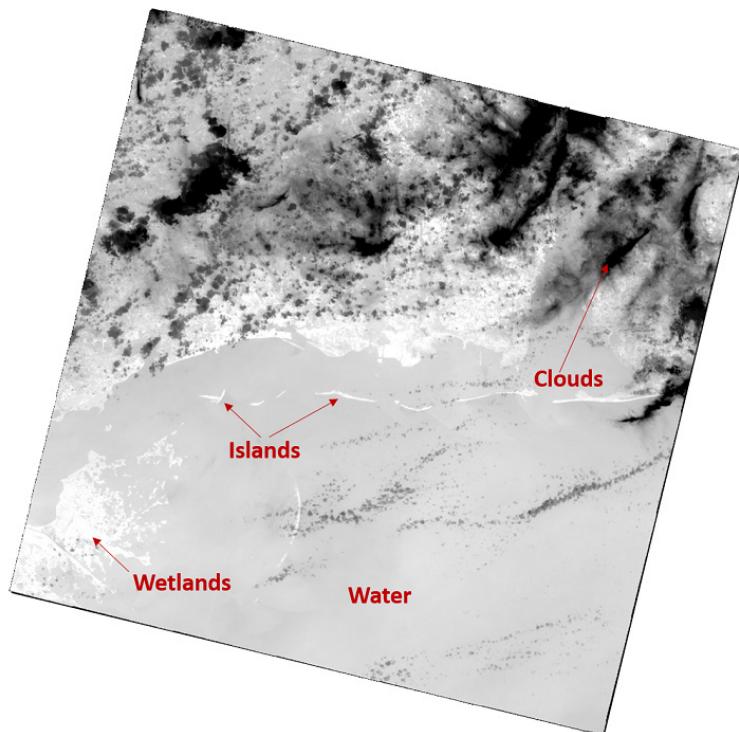


Figure 7.9 – Key features in a thermal satellite image

Because every pixel in that image has a reflectance value, it is information as opposed to just color. The type of reflectance can tell us definitively what a feature is, as opposed to us guessing by looking at it. Python can see those values and pick out features the same way we intuitively do by grouping related pixel values. We can colorize pixels based on their relation to each other to simplify the image and view-related features. This technique is called **classification**.

Classifying can range from fairly simple groupings, based only on some value distribution algorithm derived from the histogram, to complex methods involving training datasets and even computer learning and artificial intelligence. The simplest forms are called **unsupervised classifications**, in which no additional input is given other than the image itself. Methods involving some sort of training data to guide the computer are called **supervised classifications**. It should be noted that classification techniques are used across many fields, from medical doctors searching for cancerous cells in a patient's body scan to casinos using facial-recognition software on security videos to automatically spot known con artists at casino blackjack tables.

To introduce remote sensing classification, we'll just use the histogram to group pixels with similar colors and intensities and see what we get. First, you'll need to download the Landsat 8 scene from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_07\\_Asset\\_Files/thermal.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_07_Asset_Files/thermal.zip).

Instead of our `histogram()` function from the previous examples, we'll use the version included with NumPy that allows you to easily specify the number of bins and returns two arrays with the frequency, as well as the ranges of the bin values. We'll use the second array with the ranges as our class definitions for the image. The `lut` (or lookup) table is an arbitrary color palette that's used to assign colors to the 20 unsupervised classes. You can use any colors you want. Let's look at the following steps:

1. First, let's import our libraries:

```
from osgeo import gdal_array
```

2. Next, set up some variables for our input and output images:

```
src = "thermal.tif"  
tgt = "classified.jpg"
```

3. Load the image into a NumPy array for processing:

```
srcArr = gdal_array.LoadFile(src)
```

4. Now, we're going to create a histogram of our image with 20 groups, or bins, that we'll use for classifying:

```
classes = gdal_array.numpy.histogram(srcArr, bins=20) [1]
```

5. Then, create a lookup table that will define the color ranges for our classes so that we can visualize them:

```
lut = [[255, 0, 0], [191, 48, 48], [166, 0, 0], \
[255, 64, 64], [255, 115, 115], [255, 116, 0], \
[191, 113, 48], [255, 178, 115], [0, 153, 153], \
[29, 115, 115], [0, 99, 99], [166, 75, 0], \
[0, 204, 0], [51, 204, 204], [255, 150, 64], \
[92, 204, 204], [38, 153, 38], [0, 133, 0], \
[57, 230, 57], [103, 230, 103], [184, 138, 0]]
```

6. Now that our setup is complete, we can perform the classification:

```
start = 1
rgb = gdal_array.numpy.zeros((3, \
srcArr.shape[0], srcArr.shape[1], ), \
gdal_array.numpy.float32)
for i in range(len(classes)):
    mask = gdal_array.numpy.logical_and(\ 
        start <= srcArr, srcArr <= classes[i])
    for j in range(len(lut[i])):
        rgb[j] = gdal_array.numpy.choose(mask, \
            (rgb[j], lut[i][j]))
    start = classes[i]+1
```

7. Finally, we can save our classified image:

```
output = gdal_array.SaveArray(\ 
    rgb.astype(gdal_array.numpy.uint8), tgt, format="JPEG")
output = None
```

The following figure shows our classification output, which we just saved as a JPEG:



Figure 7.10 – The satellite image classified with Python

We didn't specify the prototype argument when saving this as an image, so it has no georeferencing information, though we could easily have done otherwise to save the output as a GeoTIFF.

This result isn't bad for a very simple unsupervised classification. The islands and coastal flats show up as different shades of green. The clouds were isolated as shades of orange and dark blues. We did have some confusion inland where the land features were colored the same as the Gulf of Mexico. We could further refine this process by defining the class ranges manually instead of just using the histogram.

Now that we can separate features in the image, let's try to extract features as vector data for inclusion in a GIS.

## Extracting features from images

The ability to classify an image leads us to another remote sensing capability. Now that you've worked with shapefiles over the last few chapters, have you ever wondered where they come from? Vector GIS data such as shapefiles are typically extracted from remotely sensed images such as the examples we've seen so far.

Extraction normally involves an analyst clicking around each object in an image and drawing the feature to save it as data. But with good remotely-sensed data and proper pre-processing, it is possible to automatically extract features from an image.

For this example, we'll take a subset of our Landsat 8 thermal image to isolate a group of barrier islands in the Gulf of Mexico. The islands appear white as the sand is hot and the cooler water appears black (you can download this image from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_07\\_Asset\\_Files/islands.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_07_Asset_Files/islands.zip)):

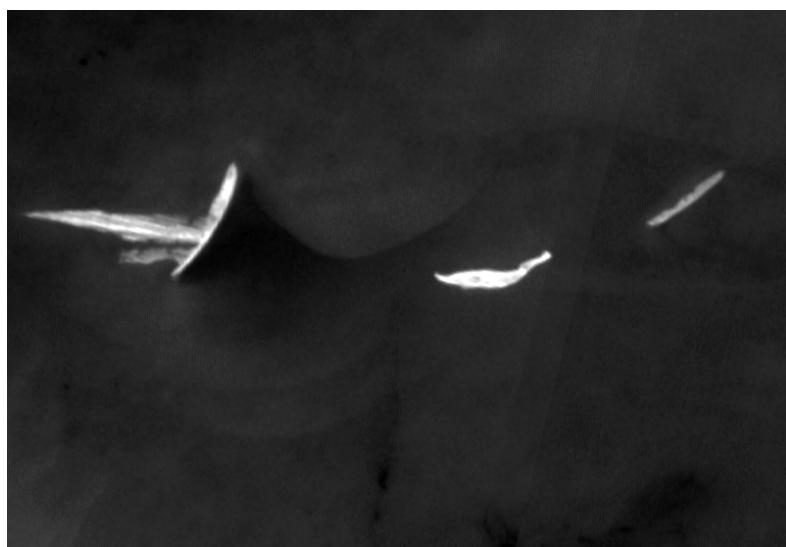


Figure 7.11 – Islands clipped out of the thermal satellite image

Our goal with this example is to automatically extract the three islands in the image as a shapefile. But before we can do that, we need to mask out any data we aren't interested in. For example, the water has a wide range of pixel values, as do the islands themselves. If we just want to extract the islands themselves, we need to push all the pixel values into just two bins to make the image black and white. This technique is called thresholding. The islands in the image have enough contrast with the water in the background that thresholding should isolate them nicely.

In the following script, we will read the image into an array and then histogram the image using only two bins. We will then use the colors black and white to color the two bins. This script is simply a modified version of our classification script with very limited output.

Let's look at the following steps:

1. First, let's import the one library we need:

```
from osgeo import gdal_array
```

2. Next, define the variables for our input and output image:

```
src = "islands.tif"  
tgt = "islands_classified.tiff"
```

3. Then, we can load the image:

```
srcArr = gdal_array.LoadFile(src)
```

4. Now, we can set up our simple classification scheme:

```
start = 1  
rgb = gdal_array.numpy.zeros((3, srcArr.shape[0], \  
srcArr.shape[1], ), gdal_array.numpy.float32)  
for i in range(len(classes)):  
    mask = gdal_array.numpy.logical_and(\br/>        start <= srcArr, srcArr <= classes[i])  
    for j in range(len(lut[i])):  
        rgb[j] = gdal_array.numpy.choose(\br/>            mask, (rgb[j], lut[i][j]))  
    start = classes[i]+1
```

5. Finally, save the image:

```
gdal_array.SaveArray(rgb.astype(gdal_array.numpy.uint8), \  
tgt, format="GTIFF", prototype=src)
```

The output looks great:



Figure 7.12 – The thresholded image

The islands are clearly isolated, so our extraction script will be able to identify them as polygons and save them to a shapefile. The GDAL library has a method called `Polygonize()` that does exactly that. It groups all sets of isolated pixels in an image and saves them as a feature dataset. One interesting technique we will use in this script is that we will use our input image as a mask.

The `Polygonize()` method allows you to specify a mask that will use the color black as a filter that will prevent the water from being extracted as a polygon, so we'll end up with just the islands. Another area to note in the script is that we copy the georeferencing information from our source image to our shapefile to geolocate it properly. Let's look at the following steps:

1. First, let's import our libraries:

```
from osgeo import gdal  
from osgeo import ogr, osr
```

2. Next, set up our input and output image and shapefile variables:

```
src = "islands_classified.tif"  
tgt = "extract.shp"  
tgtLayer = "extract"
```

3. Let's open our input image and get the first and only band:

```
srcDS = gdal.Open(src)  
band = srcDS.GetRasterBand(1)
```

4. Then, tell GDAL to use that band as a mask:

```
mask = band
```

5. Now, we're ready to set up our shapefile:

```
driver = ogr.GetDriverByName("ESRI Shapefile")  
shp = driver.CreateDataSource(tgt)
```

6. Then, copy our spatial reference information from the source image to the shapefile, to locate it on the Earth:

```
srs = osr.SpatialReference()  
srs.ImportFromWkt(srcDS.GetProjectionRef())  
layer = shp.CreateLayer(tgtLayer, srs=srs)
```

7. Now, we can set up our shapefile attributes:

```
fd = ogr.FieldDefn("DN", ogr.OFTInteger)  
layer.CreateField(fd)  
dst_field = 0
```

8. Finally, we can extract our polygons:

```
extract = gdal.Polygonize(band, mask, layer, \  
    dst_field, [], None)
```

The output shapefile is simply called `extract.shp`. As you may remember from *Chapter 4, Geospatial Python Toolbox*, we created a quick pure Python script using `PyShp` and `PNGCanvas` to visualize shapefiles. We'll bring that script back here so that we can look at our shapefile, but we'll add something extra to it. The largest island has a small lagoon that shows up as a hole in the polygon. To properly render it, we have to deal with parts in a shapefile record.

The previous example using that script did not do that, so we'll add that piece as we loop through the shapefile features:

1. First, let's need to import the libraries we'll need:

```
import shapefile  
import pngcanvas
```

2. Next, get the spatial information from the shapefile so that we can map coordinates to pixels:

```
xdist = r.bbox[2] - r.bbox[0]  
ydist = r.bbox[3] - r.bbox[1]  
iwidth = 800  
iheight = 600  
xratio = iwidth/xdist  
yratio = iheight/ydist
```

3. Now, create a list to hold our polygons:

```
polygons = []
```

4. Then, loop through the shapefile and collect our polygons:

```
for shape in r.shapes():  
    for i in range(len(shape.parts)):  
        pixels = []  
        pt = None  
        if i < len(shape.parts)-1:  
            pt = shape.points[\  
                shape.parts[i]:shape.parts[i+1]]  
        else:  
            pt = shape.points[shape.parts[i]:]  
        for x, y in pt:  
            px = int(iwidth - ((r.bbox[2] - x) * xratio))  
            py = int((r.bbox[3] - y) * yratio)  
            pixels.append([px, py])  
    polygons.append(pixels)
```

5. Next, draw the image using our polygon pixel information in PNGCanvas:

```
c = pngcanvas.PNGCanvas(iwidth, iheight)
for p in polygons:
    c.polyline(p)
with open("extract.png", "wb") as f:
    f.write(c.dump())
    f.close()
```

The following figure shows our automatically extracted island features:



Figure 7.13 – The extracted islands

Commercial packages that do this kind of work can easily cost tens of thousands of dollars. While these packages are very robust, it is still fun and empowering to see how far you can get with simple Python scripts and a few open source packages. In many cases, you can do everything you need to do.

The western-most island contains the polygon hole, as shown in the following figure, and is zoomed in on that area:

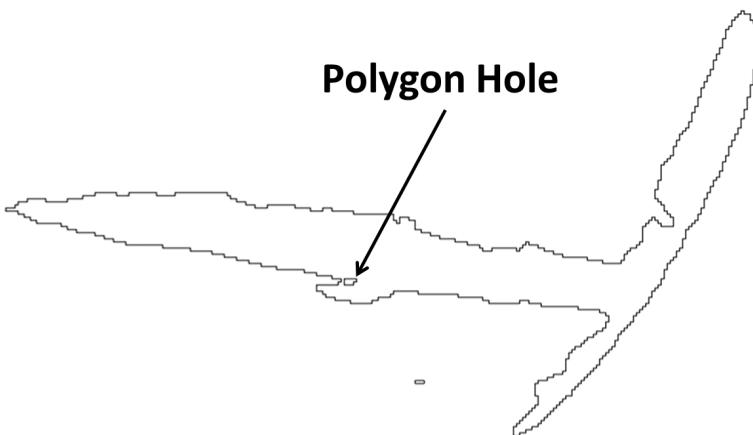


Figure 7.14 – A hole in the polygon

### Ignoring polygon holes

If you want to see what would happen if we didn't deal with the polygon holes, then just run the version of the script from *Chapter 4, Geospatial Python Toolbox*, on this same shapefile to compare the difference. The lagoon is not easy to see, but you will find it if you use the other script.

Automated feature extraction is a holy grail within geospatial analysis because of the cost and tedious effort required to manually extract features. The key to feature extraction is proper image classification. Automated feature extraction works well with water bodies, islands, roads, farm fields, buildings, and other features that tend to have high-contrast pixel values within their background.

Now that you have a good grasp of working with remote sensing data using GDAL, NumPy, and PIL, it's time to move on to our most complex example: change detection.

## Understanding change detection

Change detection is the process of taking two geo-registered images of the same area from two different dates and automatically identifying differences. It is just another form of image classification. Just like our previous classification examples, it can range from trivial techniques like those used here to highly sophisticated algorithms that provide amazingly precise and accurate results.

For this example, we'll use two images from a coastal area. These images show a populated area before and after a major hurricane, so there are significant differences, many of which are easy to visually spot, making these samples good for learning change detection. Our technique is to simply subtract the first image from the second to get a simple image difference using NumPy. This is a valid and often-used technique.

The advantages are that it is comprehensive and very reliable. The disadvantage of this overly simple algorithm is that it doesn't isolate the type of change. Many changes are insignificant for analysis, such as the waves in the ocean. In this example, we'll mask the water fairly effectively to avoid that distraction and only focus on the higher reflectance values toward the right-hand side of the difference image histogram.

### Downloading the *before* and *after* images

You can download the baseline image from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_07\\_Asset\\_Files/before.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_07_Asset_Files/before.zip). You can download the changed image from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_07\\_Asset\\_Files/after.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_07_Asset_Files/after.zip). Note these images are quite large – 24 MB and 64 MB, respectively!

The baseline image is panchromatic, while the changed image is in false color. Panchromatic images are created by sensors that capture all visible light and are typical of higher resolution sensors rather than multispectral sensors that capture bands containing restricted wavelengths.

The following is a screenshot of the baseline image:



Figure 7.15 – The baseline image

To view these images up-close yourself, you should use QGIS, as described in the *Quantum GIS* section in *Chapter 3, The Geospatial Technology Landscape*, to view them easily. The following figure shows the after image:

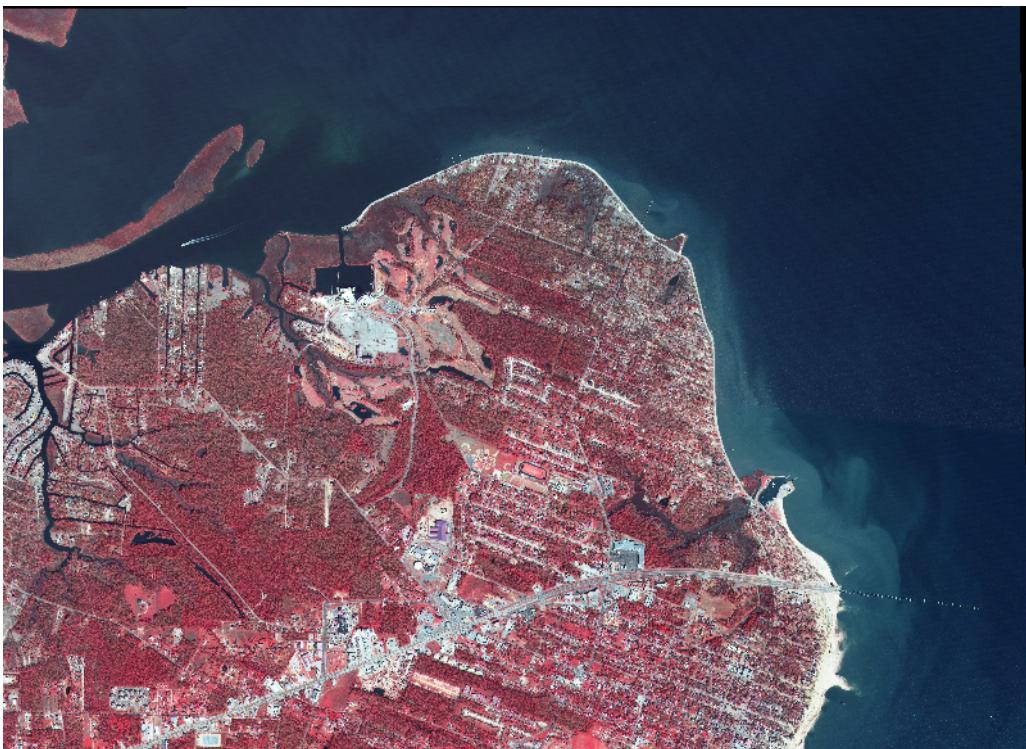


Figure 7.16 – The after image

Normally, you would use two identical band combinations, but these samples will work for our purposes. The visual markers we can use to evaluate change detection include a bridge in the southeast quadrant of the image that spans from the Peninsula to the edge of the image. This bridge is visible in the before image and is reduced to pilings by the hurricane. Another marker is a boat in the northwest quadrant, which appears in the after image as a white trail but is not in the before image.

A neutral marker is the water and the state highway, which runs through the town and connects to the bridge. This feature is easily visible concrete, which does not change significantly between the two images.

So, let's perform change detection:

1. First, let's load our libraries:

```
from osgeo import gdal_array  
import numpy as np
```

2. Now, set up the variables for our input and output images:

```
im1 = "before.tif"  
im2 = "after.tif"
```

3. Next, read both images into NumPy arrays with `gdal_array`:

```
ar1 = gdal_array.LoadFile(im1).astype(np.int8)  
ar2 = gdal_array.LoadFile(im2)[1].astype(np.int8)
```

4. Now, subtract the before image from the after image (difference = after - before):

```
diff = ar2 - ar1
```

5. Then, divide the image into five classes:

```
classes = np.histogram(diff, bins=5)[1]
```

6. Next, set our color table to use black to mask the lower classes. We're doing this to filter water and roads because they are darker in the image:

```
lut = [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], \  
[0, 255, 0], [255, 0, 0]]
```

7. Then, assign colors to the classes:

```
start = 1  
rgb = np.zeros((3, diff.shape[0], \  
diff.shape[1], ), np.int8)  
for i in range(len(classes)):  
    mask = np.logical_and(\  
        start <= diff, diff <= classes[i])  
    for j in range(len(lut[i])):  
        rgb[j] = np.choose(mask, (rgb[j], lut[i][j]))  
    start = classes[i]+1
```

8. Finally, save our image:

```
output = gdal_array.SaveArray(rgb, "change.tif", \  
format="GTiff", prototype=im2)  
output = None
```

Here's what our initial difference image looks like:

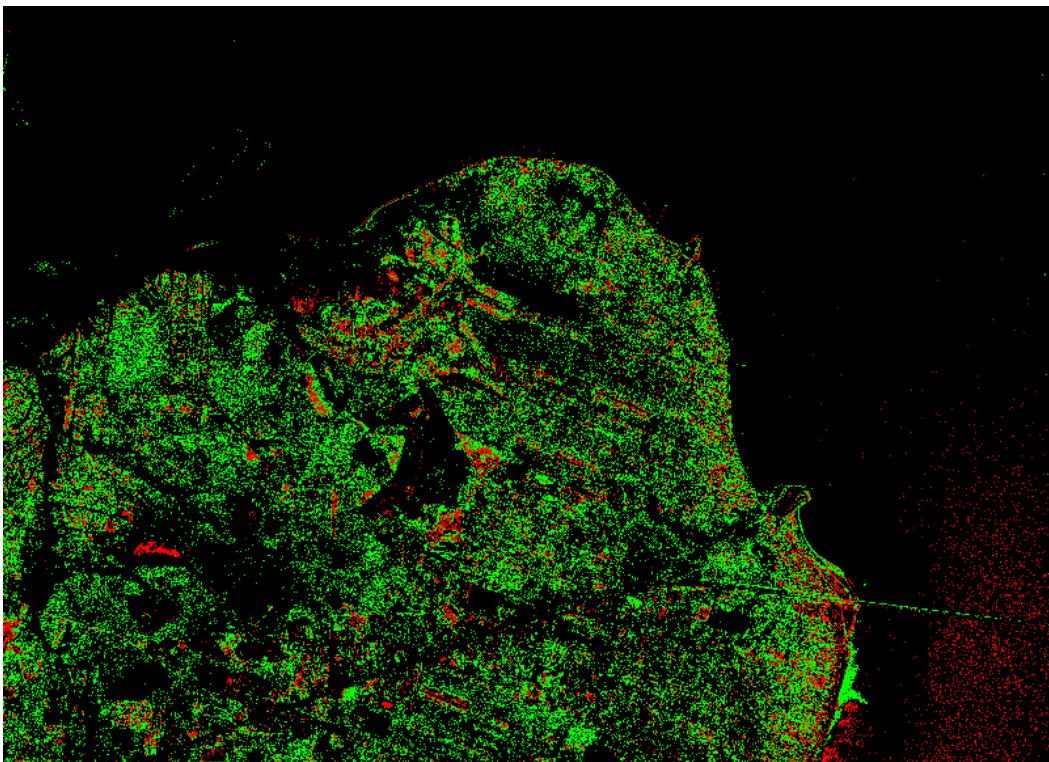


Figure 7.17 – Our colored difference image

For the most part, the green classes represent areas where something was added. The red would be a darker value where something was probably removed. We can see that the boat trail is green in the northwest quadrant. We can also see a lot of changes in vegetation, as would be expected due to seasonal differences. The bridge is an anomaly because the exposed pilings are brighter than the darker surface of the original bridge, which makes them green instead of red.

Concrete is a major indicator in change detection because it is very bright in sunlight and is usually a sign of new development. Conversely, if a building is torn down and the concrete is removed, the difference is also easy to identify. So, the simple difference algorithm that we used here isn't perfect, but it could be greatly improved using thresholding, masking, better class definitions, and other techniques.

To appreciate our change detection product, you can overlay it on the before or after image in QGIS and set the color black to transparent, as shown in the following figure:

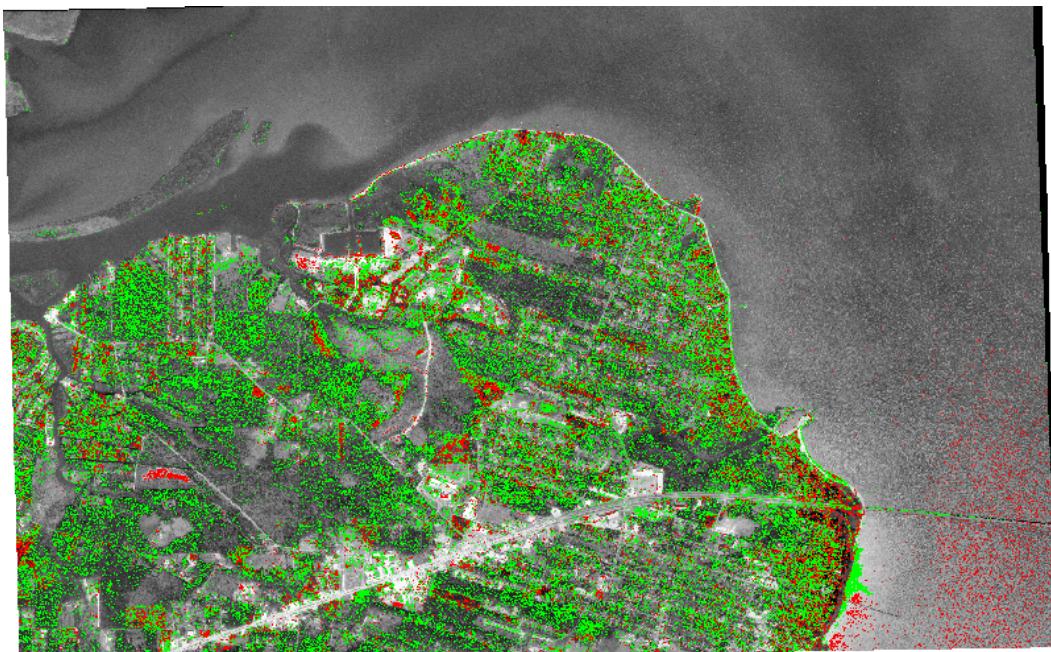


Figure 7.18 – The change detection output overlayed on the before image

Potentially, you can combine this change detection analysis with the feature extraction example to extract changes as vector data that can be analyzed in a GIS efficiently.

After diving into the intricate process of image change detection, where we identified and analyzed changes in land cover or other phenomena over time, we're ready to shift our focus to another essential task in remote sensing: extracting raster footprints as a vector file. This process involves converting the boundaries of raster data into vector format, allowing for more precise spatial analysis and integration with other vector datasets. But we've worked hard in this chapter doing complex remote sensing tasks, so in the next section, we'll ask ChatGPT to help us out with this task.

## Extracting image footprints using ChatGPT

Raster images, which are essentially grid-based digital images, provide a wealth of detailed data. However, their format can sometimes limit our ability to interact with and analyze the data they contain. This is where vector footprints come into play. By creating **vector footprints** of raster images, we essentially create an outline or footprint of the image in a vector format as polygons.

This process allows us to interact with raster data in new and more flexible ways. For instance, vector footprints can be used to quickly identify the geographic extent of raster data, which is particularly useful when dealing with large datasets. They can also be used to index raster data, making it easier

to manage and query. Furthermore, vector footprints can be overlaid on other geospatial data layers, enabling more complex spatial analyses.

For our final technique in this chapter, we'll create a footprint for the `swap.tif` image that we used in the first example. But once again, we'll ask ChatGPT to create the script for us to save time!

In a browser, go to the free ChatGPT 3.5 web interface and enter the following prompt:

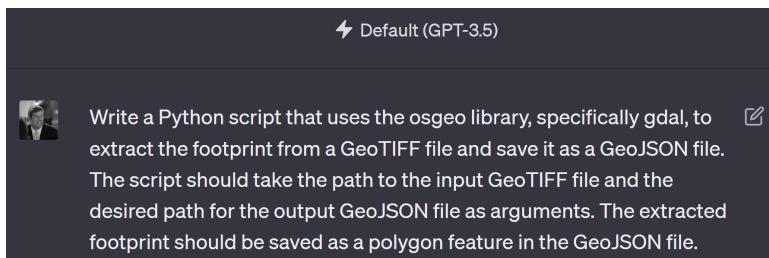


Figure 7.19 – Our ChatGPT prompt to produce a footprint script

ChatGPT will output a script similar to the following script that we'll step through. But remember that ChatGPT is unpredictable and the script it gives you might vary slightly from this one, but it should still create a working script if you use the same prompt:

1. First, it imports the necessary libraries – `gdal` for handling geospatial data and `ogr` for working with vector data:

```
from osgeo import gdal, ogr
```

2. Next, it defines the `geotiff_to_geojson(geotiff_path, geojson_path)` function, which will convert the GeoTIFF into GeoJSON:

```
def geotiff_to_geojson(geotiff_path, geojson_path):
```

3. Now, it opens the input GeoTIFF file specified by `geotiff_path`:

```
dataset = gdal.Open(geotiff_path)
```

4. Then, it gets the raster layer from the GeoTIFF file:

```
raster_layer = dataset.GetRasterBand(1)
```

5. Next, it retrieves the geotransform and size information of the raster:

```
transform = dataset.GetGeoTransform()
width = dataset.RasterXSize
height = dataset.RasterYSize
```

6. Then, it calculates the four corners of the raster based on the geotransform and size, which represent the bounding box of the raster:

```
ulx = transform[0]
uly = transform[3]
lrx = ulx + width * transform[1]
lry = uly + height * transform[5]
```

7. Next, it creates a linear ring with these four corners to define the polygon's boundary:

```
ring = ogr.Geometry(ogr.wkbLinearRing)
ring.AddPoint(ulx, uly)
ring.AddPoint(lrx, uly)
ring.AddPoint(lrx, lry)
ring.AddPoint(ulx, lry)
ring.AddPoint(ulx, uly)
```

8. After, it adds the linear ring to the polygon geometry:

```
poly = ogr.Geometry(ogr.wkbPolygon)
poly.AddGeometry(ring)
```

9. Now, it creates a new GeoJSON file:

```
driver = ogr.GetDriverByName('GeoJSON')
output_dataset = driver.CreateDataSource(geojson_path)
```

10. Finally, it writes the polygon feature representing the raster's footprint into the GeoJSON file:

```
output_layer = output_dataset.CreateLayer(\'
  'footprint', geom_type=ogr.wkbPolygon)
feature = ogr.Feature(output_layer.GetLayerDefn())
feature.SetGeometry(poly)
output_layer.CreateFeature(feature)
feature = None
output_dataset = None
dataset = None
geotiff_path = 'swap.tif'
geojson_path = 'bbox.geojson'
geotiff_to_geojson(geotiff_path, geojson_path)
```

The following figure shows our `swap.tif` image with the footprint polygon generated from the ChatGPT script overlayed on it as a dashed line:



Figure 7.20 – The raster footprint polygon

Congratulations! In addition to being a GIS analyst, you are now a budding remote sensing analyst! Let's review what we've learned.

## Summary

In this chapter, we covered the foundations of remote sensing, including metadata, band swapping, histograms, clipping images, image classification, feature extraction, change detection, and creating footprints. Like in the other chapters, we stayed as close to pure Python as possible, and where we compromised on this goal for processing speed, we limited the software libraries as much as possible to keep things simple. However, if you have the tools from this chapter installed, you have a complete remote sensing package that is limited only by your desire to learn.

The techniques in this chapter are foundational to all remote sensing processes and will allow you to build more complex operations.

In the next chapter, we'll investigate elevation data. Elevation data doesn't fit squarely in GIS or remote sensing as it has elements of both types of processing.

## Questions

1. What type of visualization shows the statistical frequency of data distribution within a dataset?
2. In remote sensing, what is the technique called for colorizing pixels based on their relation to each other?
3. What is the technique called for creating vector shapes from features in a raster image?

## Further reading

- The authors of GDAL have a set of Python examples that cover several advanced topics that may be of interest to you. You can find them at [https://github.com/OSGeo/gdal/tree/master/swig/python/gdal-utils/osgeo\\_utils](https://github.com/OSGeo/gdal/tree/master/swig/python/gdal-utils/osgeo_utils).

# 8

## Python and Elevation Data

Elevation data is one of the most fascinating types of geospatial data. It represents many different types of data sources and formats. It can display properties of both vector and raster data, resulting in unique data products. Elevation data can be used for terrain visualization, land cover classification, hydrology modeling, transportation routing, feature extraction, and many other purposes.

You can't perform all of these options with both raster and vector data, but since elevation data is three-dimensional, due to it containing  $x$ ,  $y$ , and  $z$  coordinates, you can often get more out of this data than any other type.

We will cover the following topics in this chapter:

- Accessing ASCII Grid files
- Creating a shaded relief image
- Creating elevation contours
- Working with LiDAR data
- Working with bathymetry

By the end of this chapter, you will know how to read and write elevation data in both raster and vector formats and also be able to create some derivative products.

### Technical requirements

In addition to the requirements of the previous chapter, this chapter will require installing the Python `laspyp` module. It is not available through the `conda` package manager, so run the following command:

```
pip install laspy
```

We will also be using a module called `voronoi`, which you can download from the asset files for this chapter on GitHub and place in your working directory for this chapter: [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_08\\_Asset\\_Files/voronoi.py](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_08_Asset_Files/voronoi.py). The other code files are available at [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730\\_08\\_Asset\\_Files](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730_08_Asset_Files).

## Accessing ASCII Grid files

For most of this chapter, we'll use ASCII Grid files, or ASCIIGRID. These files are a type of raster data that's usually associated with elevation data. This grid format stores data as text in equal-sized square rows and columns with a simple header. Each cell in a row/column stores a single integer used to represent a given feature of the terrain, such as the elevation, slope, or flow direction. The simplicity makes it an easy-to-use and platform-independent raster format. This format was described in the *ASCII grids* section of *Chapter 2, Learning about Geospatial Data*.

Throughout this book, we've relied on GDAL, and to some extent, even PIL, to read and write geospatial raster data, including the `gdal_array` module, so that we can load raster data into NumPy arrays. ASCII Grid allows us to read and write rasters using only Python or even NumPy because it is simple plain text.

### Using simple elevation formats

As a reminder, some elevation datasets use image formats to store elevation data. Most image formats only support 8-bit values ranging from 0 to 255; however, some formats, including TIFF, can store larger values. Geospatial software can typically display these datasets; however, traditional image software and libraries usually don't. For simplicity, in this chapter, we'll mostly stick to the ASCII Grid format for data, which is both human- and machine-readable, as well as widely supported.

## Reading grids

NumPy has the ability to read the ASCII Grid format directly using its `loadtxt()` method, which is designed to read arrays from text files. The first six lines consist of the header, which is not a part of the array. The following lines are an example grid header:

```
ncols 250
nrows 250
xllcorner 277750.0
yllcorner 6122250.0
cellsize 1.0
NODATA_value -9999
```

Let's look at what each line in the preceding code contains:

- Line 1 contains the number of columns in the grid, which is synonymous with the *x* axis.
- Line 2 represents the *y* axis as a number of rows.
- Line 3 represents the *x* coordinate of the lower-left corner, which is the minimum *x* value in meters.
- Line 4 is the corresponding minimum *y* value in the lower-left corner of the grid.
- Line 5 is the cell size or resolution of the raster. As the cells are square, only one size value is needed, as opposed to the separate *x* and *y* resolution values in most geospatial rasters.
- Line 6 is a nodata value, which is a number that's assigned to any cell for which a value is not provided.

Geospatial software ignores these cells for calculations and often allows special display settings for it, such as making them black or transparent. The -9999 value is a common no-data placeholder value that's used in the industry and is easy to detect in software but can be arbitrarily selected. Elevation with negative values (that is, bathymetry) may have valid data at -9999 meters, for instance, and may select 9999 or other values. As long as this value is defined in the header, most software will have no issues. In some examples, we'll use the number 0; however, 0 can often also be a valid data value.

The `numpy.loadtxt()` method includes an argument called `skiprows`, which allows you to specify the number of lines in the file to be skipped before reading the array values.

Let's test that out with an example ASCII Grid file that you can download from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_08\\_Asset\\_Files/myGrid.asc](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_08_Asset_Files/myGrid.asc).

Because it is a text file, it may open in your browser, so use the **File** menu in your browser to save it as `myGrid.asc` in your working directory.

To use the `numpy.loadtxt()` function, we would execute the following code:

```
import numpy
myArray = numpy.loadtxt("myGrid.asc", skiprows=6)
print(myArray)
```

This code will skip over the header and just give us the array data that contains the number 1 in all fields:

```
[[1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]  
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]  
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

This code works well, but there's one problem. NumPy allows us to skip the header but not keep it. We need to keep this so that we have a spatial reference for our data. We will also use it to save this grid or create a new one, which is something we may want to do if we need the georeferencing information or want to copy it to make a similar grid file.

To solve this problem, we'll use Python's built-in `linecache` module to grab the header. We could open the file, loop through the lines, store each one in a variable, and then close the file. However, `linecache` reduces the solution to a single line. The following line reads the first line in the file to a variable called `line1`:

```
import linecache  
line1 = linecache.getline("myGrid.asc", 1)  
print(line1)
```

The result will be the number of columns contained in the grid:

```
ncols      10
```

In the examples in this chapter, we'll use this technique to create a simple header processor that can parse these headers into Python variables in just a few lines. Now that we know how to read grids, let's learn how to write them.

## Writing grids

Writing grids in NumPy is just as easy as reading them. We use the corresponding `numpy.savetxt()` function to save a grid to a text file. The only catch is that we must build and add the six lines of header information before we dump the array to the file. In the following steps we'll save an array called `myArray` to an ASCIIGRID file called `my10x10Grid.asc`:

1. First, we'll import NumPy:

```
import numpy
```

2. Next, we'll create our example array using NumPy's `ones()` function, which fills the length and width of the grid you specify with the number 1. We'll specify a 10-by-10 grid:

```
myArray = numpy.ones((10,10))
```

3. Then we'll build our header by grabbing the dimensions of the array dynamically, but we'll hardcode the corner coordinates in this case:

```
header = "ncols {}\n".format(myArray.shape[1])
header += "nrows {}\n".format(myArray.shape[0])
header += "xllcorner 277750.0\n"
header += "yllcorner 6122250.0\n"
header += "cellsize 1.0\n"
header += "NODATA_value -9999"
```

4. Finally, we use the `savetxt()` function to save the grid to a text file specifying the header as an argument:

```
numpy.savetxt("my10x10Grid.asc", myArray, header=header,
fmt="%1.2f")
```

We make use of Python format strings, which allow you to put placeholders in a string to format the Python objects to be inserted. The `{ }` format variable turns the object you refer to into a string. In this case, we are referencing the number of columns and rows in the array.

In NumPy, an array has two properties:

- **Size:** This returns an integer for the number of values in the array
- **Shape:** This returns a tuple with the number of rows and columns, respectively

So, in the preceding example, we use the `shape` property tuple to add the row and column counts to the header of our ASCII Grid. Notice that we also add a trailing newline character for each line (`\n`). There is no reason to change the *x* and *y* values, cell size, or no data value unless we altered them in the script.

The `savetxt()` method also has an `fmt` argument, which allows you to use Python format strings to specify how the array values are written. In this case, the `%1.2f` value specifies floats with at least one number and no more than two decimal places.

As you'll see in the upcoming examples, this ability to produce valid geospatial data files using only NumPy is quite powerful. In the next couple of examples, we'll be using an ASCIIGRID **Digital Elevation Model (DEM)** of a mountainous area near Vancouver, British Columbia, in Canada. You can download it from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_08\\_Asset\\_Files/dem.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_08_Asset_Files/dem.zip).

The following image is the raw DEM that was colorized using QGIS with a color ramp, which makes the lower elevation values dark blue and higher elevation values bright red:

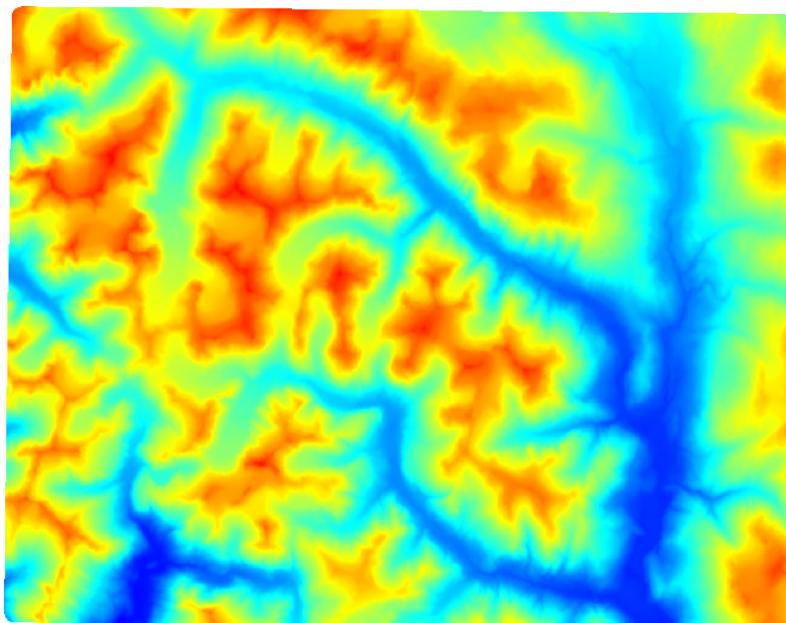


Figure 8.1 – A colorized DEM

While we can conceptually understand the data in this way, it is not an intuitive way to visualize the data. Let's see whether we can do better by creating a shaded relief.

## Creating a shaded relief

Shaded relief maps color elevation in such a way that it looks as if the terrain is cast in low-angle light, which creates bright spots and shadows. This aesthetic styling creates an almost photographic illusion, which is easy to grasp to allow us to understand the variation in the terrain. It is important to note that this style is truly an illusion as the light is often physically inaccurate in terms of the solar angle, and the elevation is usually exaggerated to increase contrast.

In this example, we'll use the ASCII DEM we referenced previously to create another grid that represents a shaded relief version of the terrain in NumPy. This terrain is quite dynamic, so we won't need to exaggerate the elevation; however, the script has a variable called `z`, which can be increased from 1.0 to scale the elevation up.

After we have defined all the variables, including the input and output filenames, we'll see the header parser based on the `linecache` module, which also uses a Python list comprehension to loop and parse the lines that are then split from a list into six variables. We also create a `y` cell size called `ycell`, which is just the inverse of the cell size by convention. If we don't do this, the resulting grid will be transposed.

### Creating slope and aspect grids

Note that we define filenames for slope and aspect grids, which are two intermediate products that are combined to create the final product. These intermediate grids are output as well. They can also serve as inputs to other types of products.

This script uses a three-by-three windowing method to scan the image and smooth out the center value in these mini-grids to process the image efficiently. It does so within the memory constraints of your computer. However, because we are using NumPy, we can process the entire array at once via matrices, as opposed to using a lengthy series of nested loops. This technique is based on the excellent work of a developer named Michal Migurski, who implemented the clever NumPy version of Matthew Perry's C++ implementation, which served as the basis for the DEM tools in the GDAL suite.

After the slope and aspect have been calculated, they are used to output the shaded relief. The slope is the steepness of a hill or mountain, while the aspect is the direction the grid cell faces that is specified as a degree between 0 and 360. Finally, everything is saved to the disk from NumPy. In the `savetxt()` method, we specify a four-integer format string as the peak elevations are several thousand meters. Let's look at the steps involved:

1. First, we'll import the `linecache` module to parse the header and the `numpy` module to do the processing:

```
from linecache import getline
import numpy as np
```

2. Next, we'll set up all of the variable names that will define how the shaded relief is processed:

```
source = "dem.asc"
slopegrid = "slope.asc"
aspectgrid = "aspect.asc"
shadegrid = "relief.asc"
azimuth = 315.0
altitude = 45.0
z = 1.0
scale = 1.0
NODATA = -9999
deg2rad = 3.141592653589793 / 180.0
rad2deg = 180.0 / 3.141592653589793
```

3. Now that our variables are set up, we can parse the header:

```
hdr = [getline(source, i) for i in range(1, 7)]
values = [float(h.split(" ")[-1].strip()) for h in hdr]
cols, rows, lx, ly, cell, nd = values
xres = cell
yres = cell * -1
```

4. Next, we can load the actual data using numpy by skipping the header portion:

```
arr = np.loadtxt(source, skiprows=6)
```

5. We're going to loop through the data, row by row, column by column, to process it. Please note, however, that we're going to skip the outer edges that contain nodata values. We'll break the data into smaller grids of 3x3 pixels as we go because for each grid cell, we need to see the cells surrounding it:

```
window = []
for row in range(3):
    for col in range(3):
        window.append(arr[row:(row + arr.shape[0] - 2),
                          col:(col + arr.shape[1] - 2)])
x = ((z * window[0] + z * window[3] + z * window[3] + \
      z * window[6]) -
      (z * window[2] + z * window[5] + z * window[5] + \
      z * window[8])) / (8.0 * xres * scale)
y = ((z * window[6] + z * window[7] + z * window[7] + \
      z * window[8]) - (z * window[0] + z * window[1] + \
      z * window[1] + z * window[2])) / (8.0 * yres * scale)
```

6. For each 3x3 mini-window, we'll calculate the slope, aspect, and then the shaded relief value:

```
slope = 90.0 - np.arctan(np.sqrt(x * x + y * y)) * rad2deg
aspect = np.arctan2(x, y)
shaded = np.sin(alitude * deg2rad) * np.sin(slope) *
         np.cos(alitude * deg2rad) * np.cos(slope) *
         np.cos((azimuth - 90.0) * deg2rad - aspect)
```

7. Next, we need to scale each value between 0 and 255 so that it can be viewed as an image:

```
shaded = shaded * 255
```

8. Now, we have to rebuild our header since we have ignored the outer edge of the nodata values and our dataset is smaller:

```
header = "ncols    {}\n".format(shaded.shape[1])
header += "nrows    {}\n".format(shaded.shape[0])
header += "xllcorner  {}\n".format(lx + (cell * \
                                         (cols - shaded.shape[1])))
header += "yllcorner  {}\n".format.ly + (cell * \
                                         (rows - shaded.shape[0]))
header += "cellsize   {}\n".format(cell)
header += "NODATA_value  {}\n".format(NODATA)
```

9. Next, we'll set any nodata values to the chosen nodata values we set in our variables at the beginning:

```
for pane in window:  
    slope[pane == nd] = NODATA  
    aspect[pane == nd] = NODATA  
    shaded[pane == nd] = NODATA
```

10. We're going to save the slope and aspect grids separately so that we can view them later and understand how the shaded relief is created:

```
with open(aspectgrid, "wb") as f:  
    f.write(bytes(header, 'UTF-8'))  
    np.savetxt(f, aspect, fmt="%4i")  
with open(shadegrid, "wb") as f:  
    f.write(bytes(header, 'UTF-8'))  
    np.savetxt(f, shaded, fmt="%4i")
```

If we load the output shaded relief grid to QGIS and specify the styling to stretch the image to the minimum and maximum values, we will see the following image:

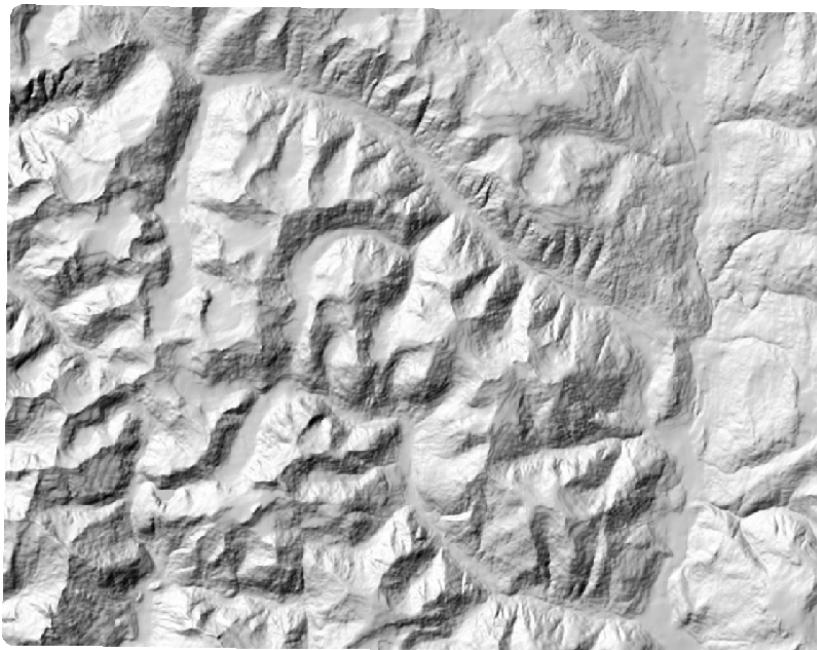


Figure 8.2 – The shaded-relief image

If QGIS asks you for a projection, the data is EPSG:3157.

As you can see, the preceding image is much easier to comprehend than the pseudo-color representation that we examined originally. Next, let's look at the slope raster that's used to create the shaded relief:



Figure 8.3 – The slope raster used to create the shaded relief

The slope shows the gradual decline in elevation from the high points to the low points in all the directions of the dataset. The slope is an especially useful input for many types of hydrology models.

Let's also look at the aspect raster. The aspect shows the maximum rate of a downslope change from one cell to its neighbors. If you compare the aspect image to the shaded relief image, you will see that the red and gray values of the aspect image correspond to shadows in the shaded relief. So, the slope is primarily responsible for turning the DEM into a terrain relief while the aspect is responsible for shading:

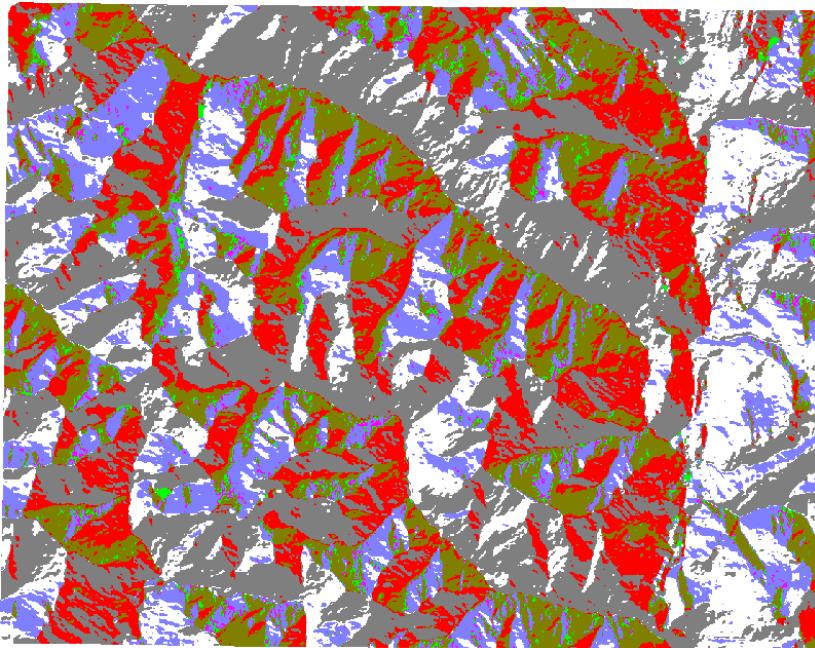


Figure 8.4 – The aspect raster used for lighting in the shaded relief

Now that we can display the data in a useful way, let's see whether we can also create other data from it.

## Creating elevation contours

A contour is an isoline along the same elevation in a dataset. Contours are usually stepped at intervals to create an intuitive way to represent elevation data, both visually and numerically, using a resource-efficient vector dataset. Now, let's look at another way to visualize the elevation better using contours.

The input is used to generate contours in our DEM and the output is a shapefile. The algorithm (*marching squares*: [https://en.wikipedia.org/wiki/Marching\\_squares](https://en.wikipedia.org/wiki/Marching_squares)) that's used to generate contours is fairly complex and very difficult to implement using NumPy's linear algebra. In this case, our solution is to fall back on the GDAL library, which has a contouring method available through the Python API. In fact, the majority of this script just sets up the OGR library code that is needed to output a shapefile. The actual contouring is a single method call named `gdal.ContourGenerate()`. Just before this call, there are comments that define the method's arguments. The most important ones are as follows:

- `contourInterval`: This is the distance in the dataset units between contours
- `contourBase`: This is the starting elevation for the contouring

- `fixedLevelCount`: This specifies a fixed number of contours as opposed to distance
- `idField`: This is the name for a required shapefile dbf field, usually just called the ID
- `elevField`: This is the name for a required shapefile dbf field for the elevation value and is useful for labeling in maps

You should have GDAL and OGR installed from the *GDAL* section of *Chapter 4, Geospatial Python Toolbox*. We will be implementing the following steps:

1. We will define the input DEM filename.
2. We will output the shapefile's name.
3. We'll create the shapefile data source with OGR.
4. We'll get the OGR layer.
5. We'll open the DEM.
6. Finally, we'll generate contours on the OGR layer.

Let's look at a code representation of the preceding steps:

1. First, we load in the `gdal` and `ogr` libraries to handle the data:

```
from osgeo import gdal, ogr
```

2. Then we'll set up a variable for our filename:

```
source = "dem.asc"
```

3. Next, we'll create the beginnings of our output shapefile using OGR:

```
target = "contour"
ogr_driver = ogr.GetDriverByName('ESRI Shapefile')
ogr_ds = ogr_driver.CreateDataSource(target + ".shp")
ogr_lyr = ogr_ds.CreateLayer(target, geom_type=ogr.
wkbLineString25D)
field_defn = ogr.FieldDefn('ID', ogr.OFTInteger)
ogr_lyr.CreateField(field_defn)
field_defn = ogr.FieldDefn('ELEV', ogr.OFTReal)
ogr_lyr.CreateField(field_defn)
```

4. Then, we'll create some contours:

```
ds = gdal.Open('dem.asc')
gdal.ContourGenerate(ds.GetRasterBand(1), 400, 10, \
[], 0, 0, ogr_lyr, 0, 1)
```

Now that we've produced a shapefile, let's draw the contour shapefile that we just created using `pngcanvas`, which we introduced in *Chapter 4, Geospatial Python Toolbox*:

1. First, we import the required libraries to handle shapefiles and create PNG canvases:

```
import shapefile  
import pngcanvas
```

2. Next, we read the shapefile named `contour.shp` to work with the contour data:

```
r = shapefile.Reader("contour.shp")
```

3. We calculate the distance in the *x* and *y* directions, and set the image width and height:

```
xdist = r.bbox[2] - r.bbox[0]  
ydist = r.bbox[3] - r.bbox[1]  
iwidth = 800  
iheight = 600
```

4. We determine the ratio of the image dimensions to the *x* and *y* distances to scale the contours properly:

```
xratio = iwidth/xdist  
yratio = iheight/ydist
```

5. We loop through the shapes in the shapefile, converting the coordinates to pixel values and storing them in the `contours` list:

```
contours = []  
for shape in r.shapes():  
    for i in range(len(shape.parts)):  
        pixels = []  
        pt = None  
        if i < len(shape.parts) - 1:  
            pt = shape.points[shape.parts[i]:shape.parts[i+1]]  
        else:  
            pt = shape.points[shape.parts[i]:]  
        for x, y in pt:  
            px = int(iwidth - ((r.bbox[2] - x) * xratio))  
            py = int((r.bbox[3] - y) * yratio)  
            pixels.append([px, py])  
    contours.append(pixels)
```

6. We create a PNG canvas with the specified width and height and set the color to red:

```
canvas = pngcanvas.PNGCanvas(iwidth, iheight)
red = [0xff, 0, 0, 0xff]
canvas.color = red
```

7. We draw the contours on the canvas using the polyline method:

```
for c in contours:
    canvas.polyline(c)
```

8. Finally, we write the canvas to a PNG file named `contours.png` and close the file:

```
f = open("contours.png", "wb")
f.write(canvas.dump())
f.close()
```

The code will output the following image:

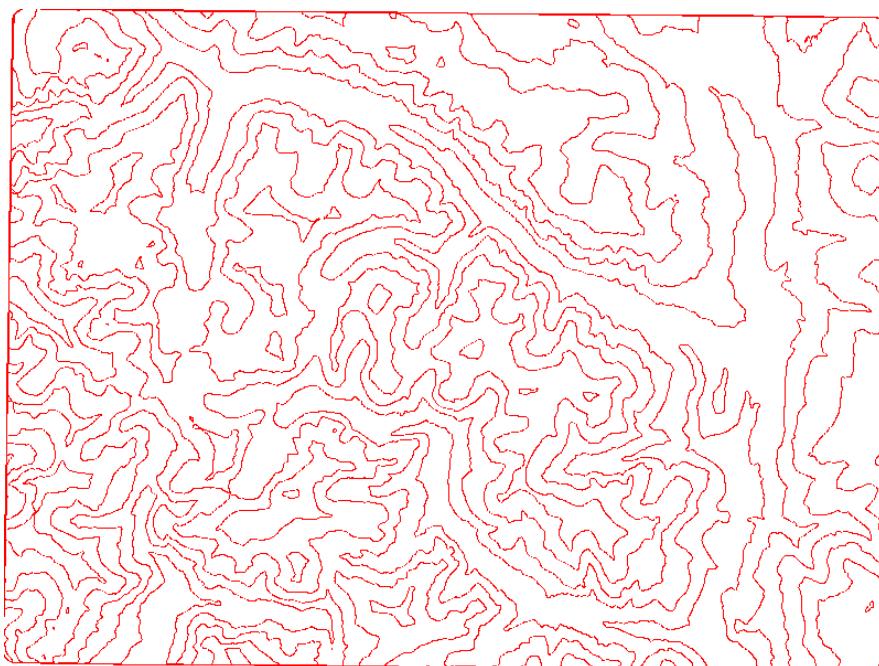


Figure 8.5 – Contour shapefile drawn with `pngcanvas`

If we bring our shaded relief ASCIIIGRID and the shapefile into a desktop GIS software, such as QGIS, we can create a simple topographic map, as follows. You can use the elevation (that is, ELEV) dbf field that you specified in the script to label the contour lines with the elevation:

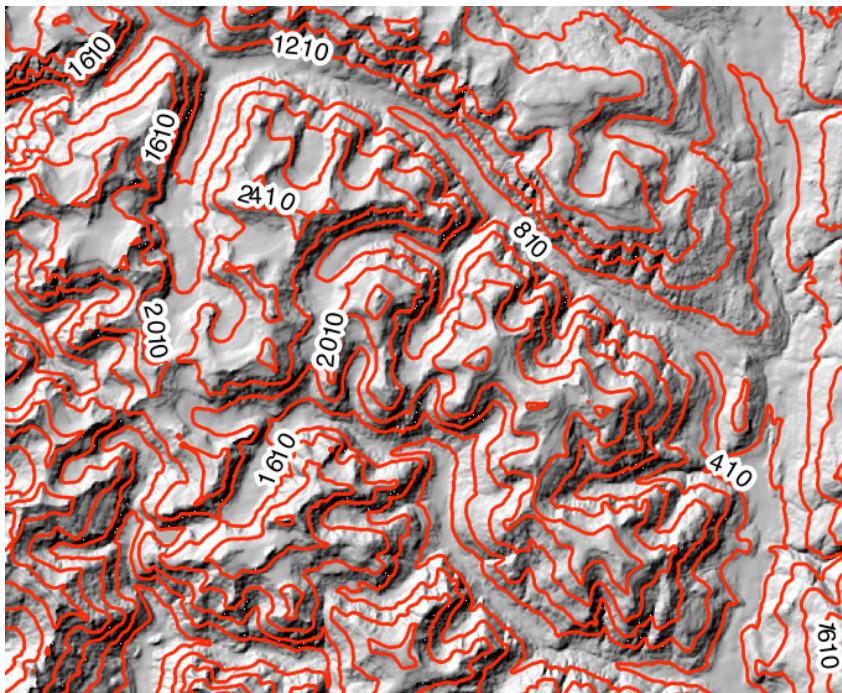


Figure 8.6 – Contour shapefile overlay on our shaded relief image

The techniques that were used in these NumPy grid examples provide the building blocks for all kinds of elevation products. Next, we'll work with one of the most complex elevation data types: LiDAR data.

## Working with LiDAR data

**LiDAR** stands for **Light Detection and Ranging**. It is similar to radar-based images but uses finite laser beams that hit the ground hundreds of thousands of times per second to collect a huge amount of very fine ( $x$ ,  $y$ ,  $z$ ) locations, as well as time and intensity. The intensity value is what really separates LiDAR from other data types. For example, the asphalt rooftop of a building may be of the same elevation as the top of a nearby tree, but the intensities will be different. Just like remote sensing, radiance values in a multispectral satellite image allow us to build classification libraries. The intensity values of LiDAR data allow us to classify and colorize LiDAR data.

The high volume and precision of LiDAR actually make it difficult to use. A LiDAR dataset is referred to as a point cloud because the shape of the dataset is usually irregular as the data is three-dimensional with outlying points. There are not many software packages that effectively visualize point clouds.

Furthermore, an irregular-shaped collection of finite points is just hard to interact with, even when we are using the appropriate software.

For these reasons, one of the most common operations on LiDAR data is to project data and resample it to a regular grid. We'll do this using a small LiDAR dataset. This dataset is approximately 7 MB uncompressed and contains over 600,000 points. The data captures some easily identifiable features, such as buildings, trees, and cars in parking lots. You can download the zipped dataset from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_08\\_Asset\\_Files/lidar.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_08_Asset_Files/lidar.zip).

The file format is a very common binary format specific to LiDAR called **LAS**, which is short for laser. Unzip this file to your working directory. To read this format, we'll use the pure Python library `laspy`. With `laspy` installed, we are ready to create a grid from LiDAR.

## Creating a grid from the LiDAR data

This script is fairly straightforward. We loop through the  $(x, y)$  point locations in the LiDAR data and project them to our grid with a cell size of 1 meter. Due to the precision of the LiDAR data, we'll end up with multiple points in a single cell. We average these points to create a common elevation value. Another issue that we have to deal with is data loss. Whenever you resample the data, you lose information.

In this case, we'll end up with nodata holes in the middle of the raster. To deal with this issue, we fill these holes with average values from the surrounding cells, which is a form of interpolation. We only need two modules, both available on PyPI, as outlined in the following code:

1. First, we import the necessary libraries to handle LAS files and perform numerical operations:

```
import laspy
import numpy as np
```

2. We define the source LAS file, target ASCII file, cell size, and a constant for nodata:

```
source = "lidar.las"
target = "lidar.asc"
cell = 1.0
NODATA = 0
```

3. We read the LAS file to work with the LiDAR data:

```
las = laspy.read(source)
```

4. We calculate the minimum and maximum coordinates and the distances in the  $x$  and  $y$  directions:

```
min = las.header.min
max = las.header.max
xdist = max[0] - min[0]
ydist = max[1] - min[1]
```

5. We calculate the number of rows and columns based on the distances and cell size:

```
cols = int(xdist) / cell
rows = int(ydist) / cell
cols = int(cols + 1)
rows = int(rows + 1)
```

6. We initialize arrays to keep track of the count and sum of  $z$  values:

```
count = np.zeros((rows, cols), dtype=float)
zsum = np.zeros((rows, cols), dtype=float)
ycell = -1 * cell
```

7. We project the  $x$  and  $y$  coordinates to the grid:

```
projx = (las.x - min[0]) / cell
projy = (las.y - min[1]) / ycell
ix = projx.astype(np.int32)
iy = projy.astype(np.int32)
```

8. We loop through the LiDAR points, updating the count and sum of  $z$  values:

```
for x, y, z in np.nditer([ix, iy, las.z]):
    count[y, x] += 1
    zsum[y, x] += z
```

9. We calculate the average  $z$  values and perform interpolation:

```
nonzero = np.where(count > 0, count, 1)
zavg = zsum / nonzero
mean = np.ones((rows, cols)) * np.mean(zavg)
left = np.roll(zavg, -1, 1)
lavg = np.where(left > 0, left, mean)
right = np.roll(zavg, 1, 1)
ravg = np.where(right > 0, right, mean)
interpolate = (lavg + ravg) / 2
fill = np.where(zavg > 0, zavg, interpolate)
```

10. We create the header for the ASCII file with the necessary metadata:

```
header = "ncols      {}\\n".format(fill.shape[1])
header += "nrows      {}\\n".format(fill.shape[0])
header += "xllcorner  {}\\n".format(min[0])
header += "yllcorner  {}\\n".format(min[1])
header += "cellsize    {}\\n".format(cell)
header += "NODATA_value {}\\n".format(NODATA)
```

11. Finally, we write the header and the filled data to the target ASCII file:

```
with open(target, "wb") as f:  
    f.write(header, 'UTF-8'))  
    np.savetxt(f, fill, fmt="%1.2f")
```

The result of our script is an ASCIIGRID, which looks like the following image when viewed in QGIS. Higher elevations are shown lighter while lower elevations are darker. Even in this form, you can see buildings, trees, and cars:

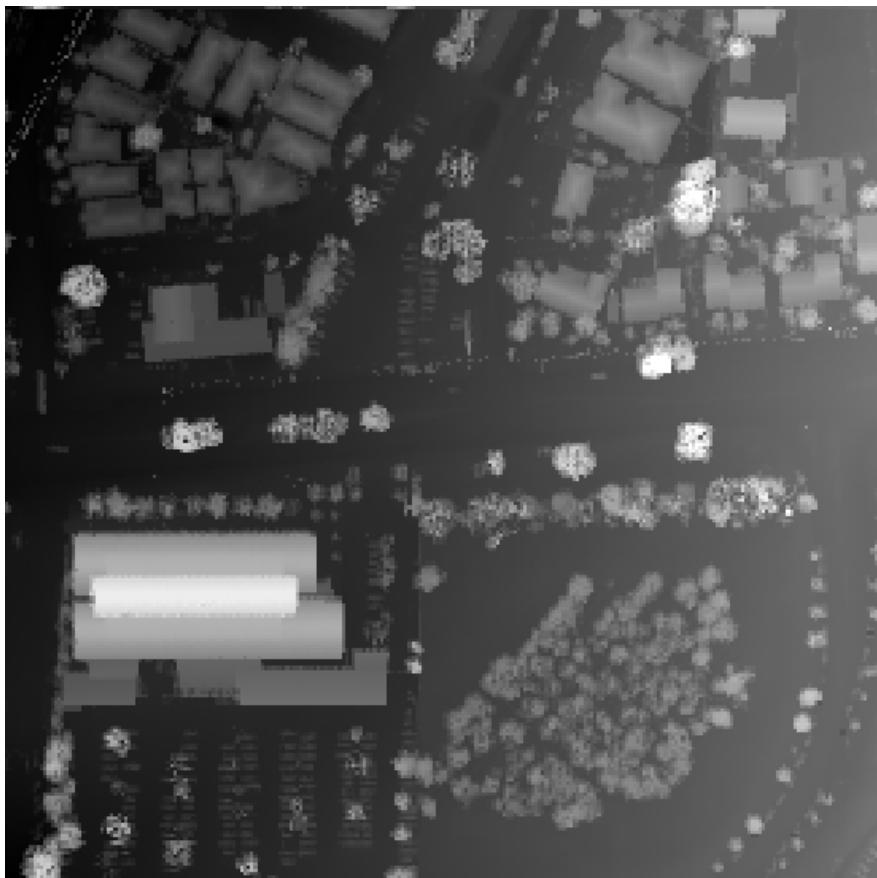


Figure 8.7 – Our LiDAR data converted to an elevation grid raster or DEM

If we assigned a heat map color ramp, the colors give you a sharper sense of the elevation differences:

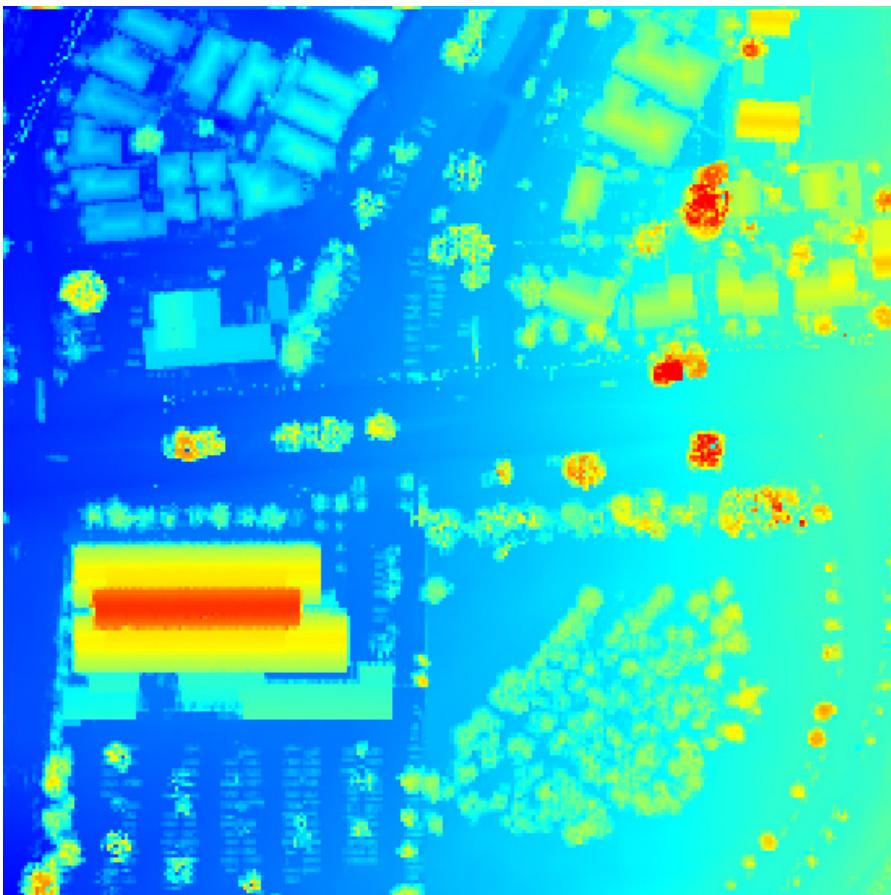


Figure 8.8 – The DEM with a color ramp based on height – red is higher and blue is lower

So, what happens if we run this output DEM through our shaded relief script from earlier? There's a big difference between straight-sided buildings and sloping mountains. If you change the input and output names in the shaded relief script to process the LiDAR DEM, we get the following slope result:

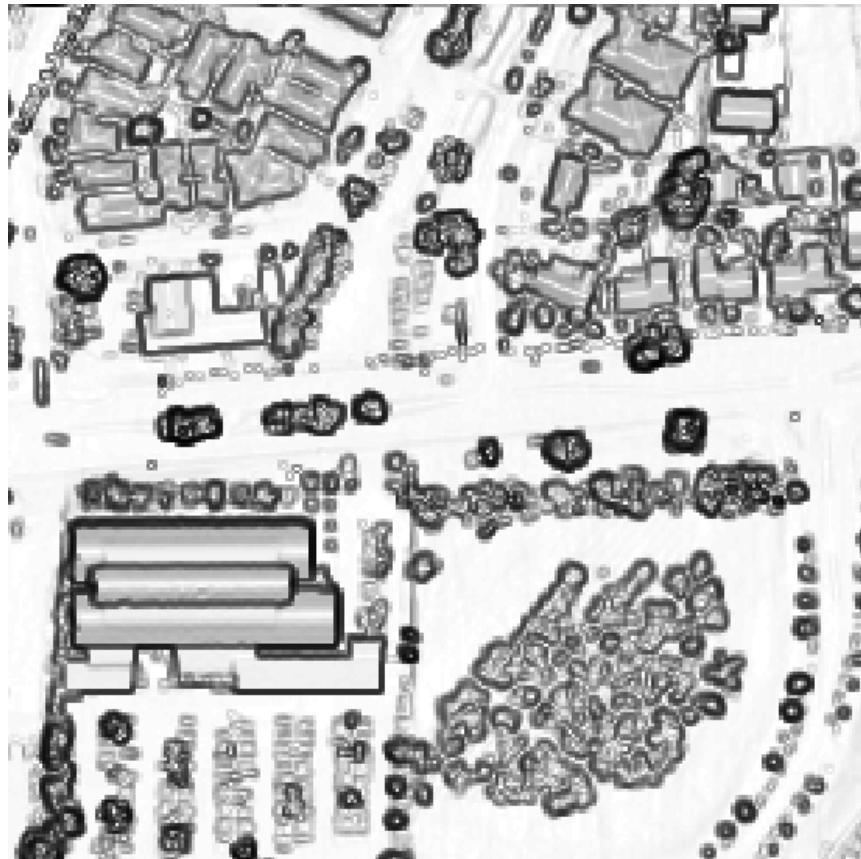


Figure 8.9 – The slope analysis of the DEM

The gently rolling slope of the mountainous terrain is reduced to outlines of major features in the image. In the aspect image, the changes are so sharp and over such short distances that the output image is very chaotic to view, as shown in the following screenshot:

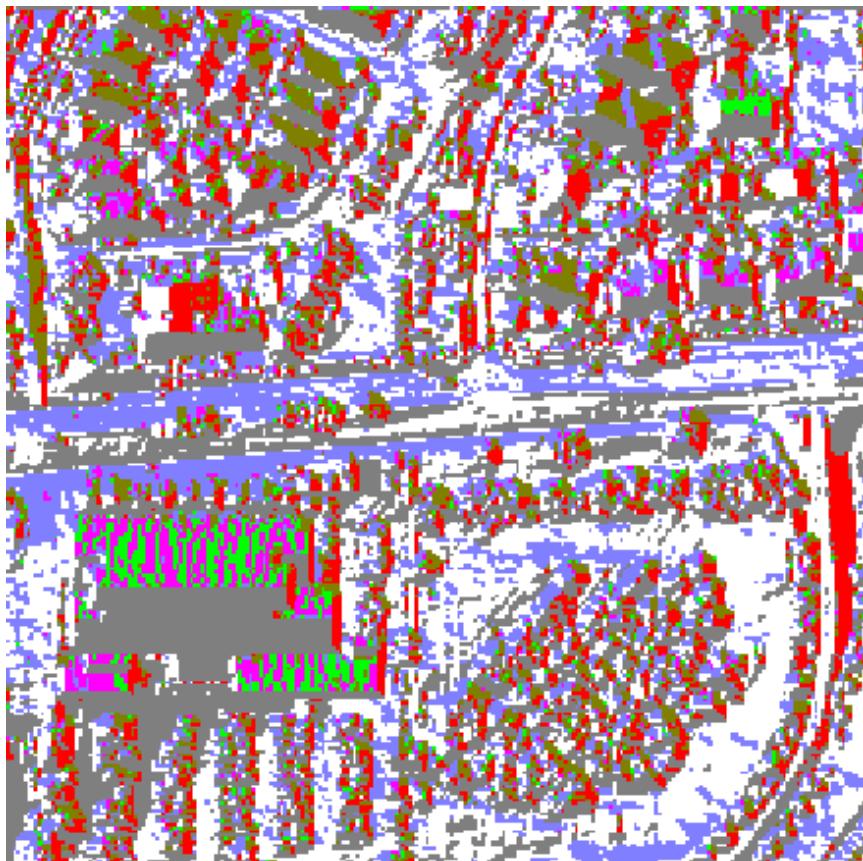


Figure 8.10 – The aspect analysis of our DEM

Despite the difference between these images and the coarser but somewhat smoother mountain versions, we still get a very nicely shaded relief, which visually resembles a black and white photograph:

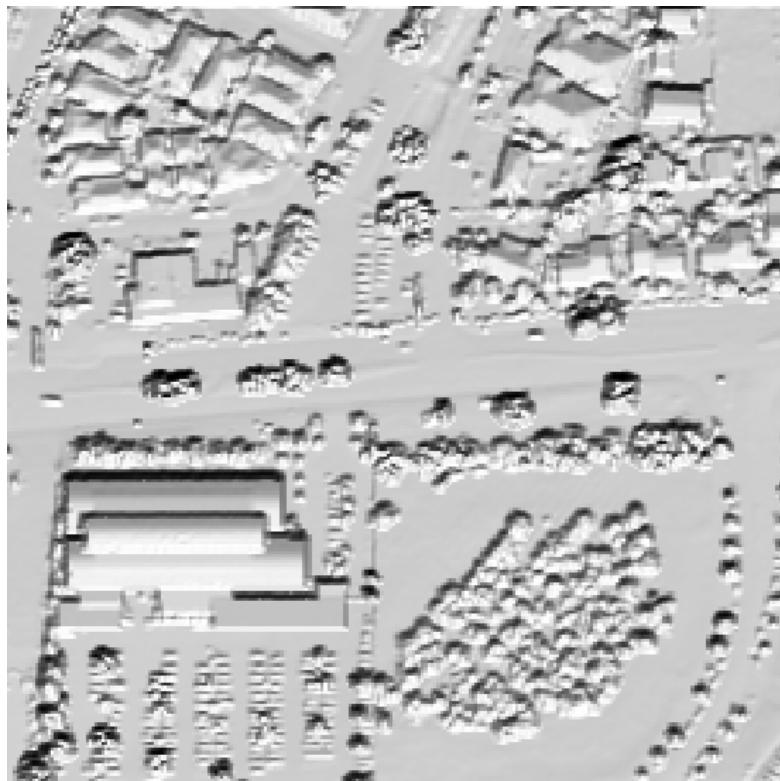


Figure 8.11 – The DEM shaded relief

Now that we know how to process LiDAR data, let's learn how to visualize it using Python.

### Using PIL to visualize LiDAR data

The previous DEM images in this chapter were visualized using QGIS and OpenEV. We can also create output images in Python by introducing some new functions of the **Python Imaging Library (PIL)** that we didn't use in the previous chapters.

In this example, we'll use the `PIL.ImageOps` module, which has functions for histogram equalization and automatic contrast enhancement. We'll use PIL's `fromarray()` method to import the data from numpy. Let's see how close we can get to the output of the desktop GIS programs that were pictured in this chapter with the help of the following code. We'll use a terrain file called `relief.asc`, which you can download from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_08\\_Asset\\_Files/relief.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_08_Asset_Files/relief.zip).

Once you're ready, let's step through the code:

1. First, we import the necessary libraries, including numpy for numerical operations and Image and ImageOps for image processing. We include a try-except block to handle different import methods for the Image library:

```
try:  
    import Image  
    import ImageOps  
except:  
    from PIL import Image, ImageOps
```

2. We define the source ASCII file containing the LiDAR DEM data and the target image file where the result will be saved:

```
source = "relief.asc"  
target = "relief.bmp"
```

3. We load the ASCII DEM file into a numpy array, skipping the first six rows, which usually contain header information:

```
arr = np.loadtxt(source, skiprows=6)
```

4. We convert the numpy array into an image using the Image library and convert it into an RGB format:

```
im = Image.fromarray(arr).convert('RGB')
```

5. We enhance the image by equalizing the histogram and automatically adjusting the contrast. This step helps in visualizing the elevation data more clearly:

```
im = ImageOps.equalize(im)  
im = ImageOps.autocontrast(im)
```

6. Finally, we save the enhanced image to the target file in BMP format:

```
im.save(target)
```

As you can see, in the following image, the enhanced shaded relief has sharper relief than the previous version:



Figure 8.12 – An enhanced shaded relief using PIL

Now, let's colorize our shaded relief. We'll use the built-in Python `colorsys` module for color space conversion. Normally, we specify colors as RGB values. However, to create a color ramp for a heat map scheme, we'll use **Hue, Saturation, and Value (HSV)** values to generate our colors.

The advantage of HSV is that you can tweak the H value to be a degree between 0 and 360 on a color wheel. Using a single value for hue allows you to use a linear ramping equation, which is much easier than trying to deal with combinations of three separate RGB values. The following diagram from Wikipedia illustrates the concepts of the HSV color model:

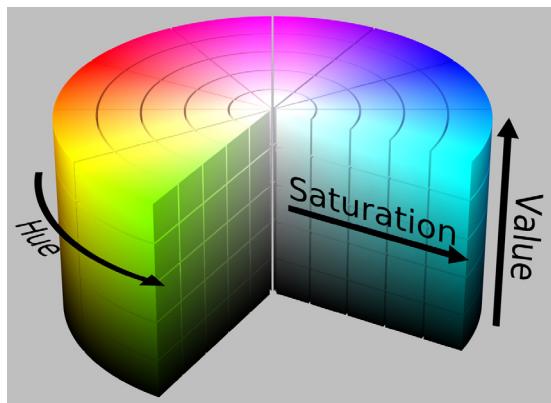


Figure 8.13 – The HSV color model (source: [https://en.wikipedia.org/wiki/HSL\\_and\\_HSV](https://en.wikipedia.org/wiki/HSL_and_HSV))

The `colorsys` module lets you switch back and forth between the HSV and RGB values. The module returns percentages for RGB values, which must then be mapped to the 0-255 scale for each color.

In the following code, we'll convert the ASCII DEM into a PIL image, build our color palette, apply the color palette to the grayscale image, and save the image:

1. First, we import the necessary libraries, including `numpy` for numerical operations, `Image` and `ImageOps` for image processing, and `colorsys` for color space conversion:

```
import numpy as np
try:
    import Image
    import ImageOps
except:
    from PIL import Image, ImageOps
import colorsys
```

2. We define the source ASCII file containing the LiDAR DEM data and the target image file where the result will be saved:

```
source = "lidar.asc"
target = "lidar.bmp"
```

3. We load the ASCII DEM file into a `numpy` array, skipping the first six rows, which usually contain header information:

```
arr = np.loadtxt(source, skiprows=6)
```

4. We convert the `numpy` array into an image using the `Image` library and convert it into a grayscale ('L') format:

```
im = Image.fromarray(arr).convert('L')
```

5. We enhance the image by equalizing the histogram and automatically adjusting the contrast:

```
im = ImageOps.equalize(im)
im = ImageOps.autocontrast(im)
```

6. We begin building our color ramp by defining the HSV color space and stepping through the colors from blue to red:

```
palette = []
h = .67
s = 1
v = 1
step = h / 256.0
for i in range(256):
```

```
rp, gp, bp = colorsys.hsv_to_rgb(h, s, v)
r = int(rp * 255)
g = int(gp * 255)
b = int(bp * 255)
palette.extend([r, g, b])
h -= step
```

7. We apply the created palette to the image, effectively colorizing the elevation data:

```
im.putpalette(palette)
```

8. Finally, we save the colorized image to the target file in BMP format:

```
im.save(target)
```

The preceding code produces the following image, with higher elevations in warmer colors and lower elevations in cooler colors:

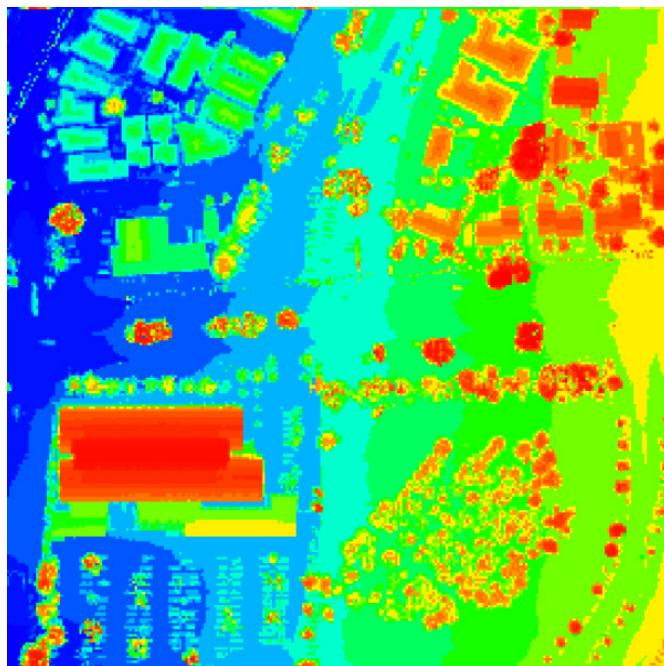


Figure 8.14 – Colorized LiDAR image using warmer colors for higher elevations

In this image, we actually get more variation than the default QGIS version. We could potentially improve this image with a smoothing algorithm that would blend the colors where they meet and soften the image visually. As you can see, we have the full range of our color ramp expressed from cool to warm colors, as the elevation change increases.

## Creating a triangulated irregular network

The following example is our most sophisticated example yet. A **triangulated irregular network (TIN)** is a vector representation of a point dataset in a vector surface of points connected as triangles. An algorithm determines which points are absolutely necessary to accurately represent the terrain as opposed to a raster, which stores a fixed number of cells over a given area and may repeat elevation values in adjacent cells that could be more efficiently stored as a polygon.

A TIN can also be resampled more efficiently on the fly than a raster, which requires less computer memory and processing power when using TIN in a GIS. The most common type of TIN is based on Delaunay triangulation, which includes all the points without redundant triangles.

The Delaunay triangulation is very complex. We'll use a pure Python library written by Bill Simons as part of Steve Fortune's Delaunay triangulation algorithm called `voronoi.py` to calculate the triangles in our LiDAR data. You can download the script to your working directory or site-packages directory from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_08\\_Asset\\_Files/voronoi.py](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_08_Asset_Files/voronoi.py).

This script reads the LAS file, generates the triangles, loops through them, and writes out a shapefile. For this example, we'll use a clipped version of our LiDAR data to reduce the area to process. If we run our entire dataset of 600,000+ points, the script will run for hours and generate over half a million triangles. You can download the clipped LiDAR dataset as a ZIP file from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_08\\_Asset\\_Files/clippedLAS.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_08_Asset_Files/clippedLAS.zip).

We have several status messages that print while the script runs because of the time-intensive nature of the following example, which can take several minutes to complete. We'll be storing the triangles as `PolygonZ` types, which allow the vertices to have a `z` elevation value. Unzip the LAS file and run the following code to generate a shapefile called `mesh.shp`:

1. First we import the necessary libraries, including `pickle` for serialization, `os` and `time` for system operations, `math` and `numpy` for mathematical functions, `shapefile` for working with shapefiles, `laspy` for handling LAS files, and `voronoi` for Delaunay triangulation:

```
import pickle
import os
import time
import math
import numpy as np
import shapefile
import laspy
import voronoi
```

2. We define the source LAS file, target shapefile, and an archive file to store the triangles:

```
source = "clippedLAS.las"
target = "mesh"
archive = "triangles.p"
```

3. We define a Point class required by the voronoi module to represent the *x* and *y* coordinates:

```
class Point:
    def __init__(self, x, y):
        self.px = x
        self.py = y
    def x(self):
        return self.px
    def y(self):
        return self.py
```

4. We read the LAS file and assemble the points into a list of Point objects:

```
las = laspy.read(source)
points = []
for x, y in np.nditer((las.x, las.y)):
    points.append(Point(x, y))
```

5. We perform Delaunay triangulation on the points and save the triangles to a pickle file:

```
triangles = voronoi.computeDelaunayTriangulation(points)
f = open(archive, "wb")
pickle.dump(triangles, f, protocol=2)
f.close()
```

6. We create a POLYGONZ shapefile and define the fields for storing the coordinates of the triangles:

```
w = shapefile.Writer(target, shapefile.POLYGONZ)
w.field("X1", "C", "40")
w.field("X2", "C", "40")
w.field("X3", "C", "40")
w.field("Y1", "C", "40")
w.field("Y2", "C", "40")
w.field("Y3", "C", "40")
w.field("Z1", "C", "40")
w.field("Z2", "C", "40")
w.field("Z3", "C", "40")
```

7. We loop through the triangles, calculate the coordinates, check for large triangles along the convex hull, and write the valid triangles to the shapefile:

```
tris = len(triangles)
last_percent = 0
for i in range(tris):
    # ... (code to calculate coordinates and check segments) ...
    part.append([x1, y1, z1, 0])
    part.append([x2, y2, z2, 0])
    part.append([x3, y3, z3, 0])
    w.polyz([part])
    w.record(x1, x2, x3, y1, y2, y3, z1, z2, z3)
```

8. Finally we save the shapefile and print a completion message:

```
print("Saving shapefile...")
w.close()
print("Done.")
```

The following image shows a zoomed-in version of the TIN over the colorized LiDAR data:

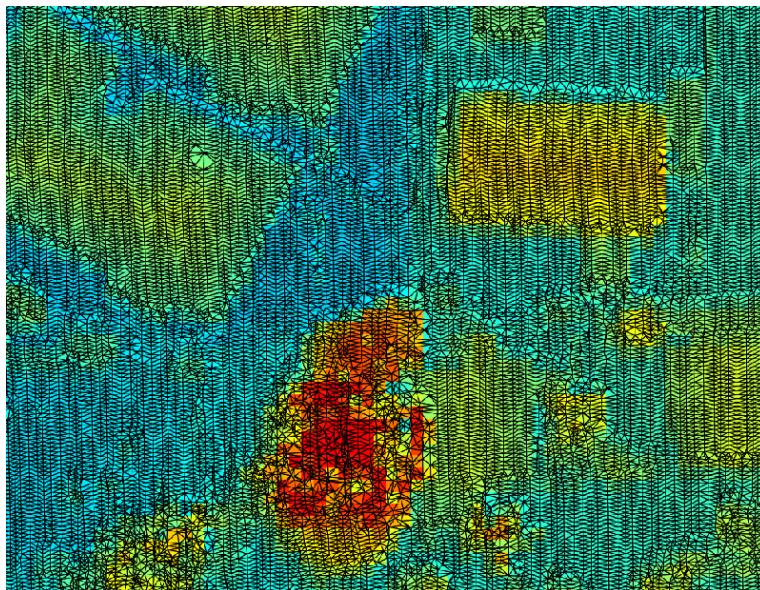


Figure 8.15 – The Voronoi mesh

The mesh provides an efficient, continuous surface from point cloud data, which is often easier to deal with than the point clouds themselves.

## Colorizing LiDAR with aerial images

LiDAR data provides a rich, three-dimensional view of the terrain, but it often lacks the visual context that can be provided by color. While coloring LiDAR points by height value as we did earlier in this chapter offers valuable insights into the topographical variations, it may not always capture the true appearance of the landscape. Utilizing an aerial photograph to colorize the LiDAR points introduces a new dimension of realism and detail, rendering the LiDAR data in colors that match the actual appearance of the Earth's surface.

The process of colorizing LiDAR data using aerial imagery is more than a visual enhancement – it's a valuable analytical tool. By aligning the LiDAR points with corresponding pixels in an aerial image, the color information from the photograph can be mapped to the LiDAR points, reflecting not only the shape but also the material properties of the landscape. This colorization technique can be instrumental in identifying features such as vegetation, roads, buildings, and water bodies. By integrating the geometric precision of LiDAR with the rich color information from aerial imagery, analysts can achieve a more comprehensive understanding of the environment, supporting applications ranging from urban planning to environmental monitoring.

To demonstrate this concept, we're going to use the following aerial image from an uninhabited island, which you can download as a GeoTiff from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_08\\_Asset\\_Files/DeerIslandAerial.tif](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_08_Asset_Files/DeerIslandAerial.tif).



Figure 8.16 – Aerial image of a portion of an uninhabited island

Then we'll use the following corresponding LiDAR file for this area, which you can download as a LAS file from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_08\\_Asset\\_Files/DeerIslandLidar.las](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_08_Asset_Files/DeerIslandLidar.las).

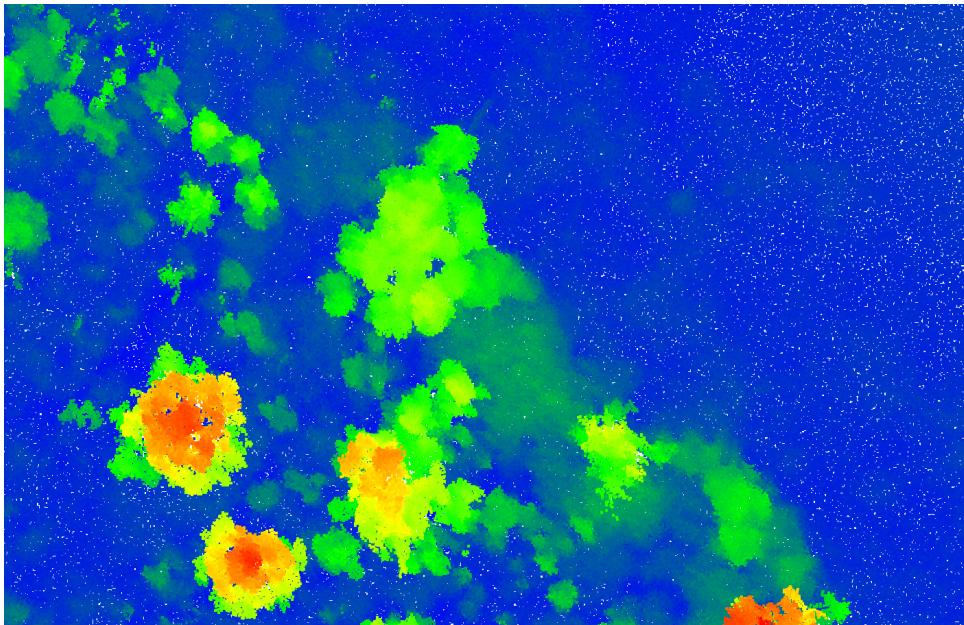


Figure 8.17 – A height-ramp-colorized version of the LiDAR file

If we look at this profile from a 90-degree angle in a 3D map view in QGIS, it looks like the following image where you can see the ground and the tree line:

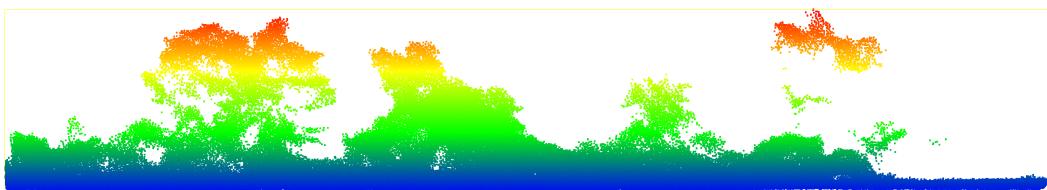


Figure 8.18 – Profile of the LiDAR data

The following code allows us to colorize a LiDAR (LAS) file using the color information from a corresponding GeoTIFF file, which can be very useful in geospatial analysis and visualization:

1. First, we import the necessary libraries to handle LAS and GeoTIFF files, as well as numerical operations with NumPy:

```
import laspy
from osgeo import gdal
import numpy as np
```

2. Next, we define the `read_geotiff` function to read a GeoTIFF file. Inside this function, we open the GeoTIFF file, extract the geotransform, and read the RGB bands:

```
def read_geotiff(geotiff_path):
    raster = gdal.Open(geotiff_path)
    geotransform = raster.GetGeoTransform()
    bands = [raster.GetRasterBand(i + 1).ReadAsArray() \
        for i in range(3)]
    return bands, geotransform
```

3. Then, we define the `get_pixel_value` function to obtain the pixel value based on the  $x$ ,  $y$  coordinates, geotransform, and bands. It also checks for invalid coordinates and returns a default value if out of bounds:

```
def get_pixel_value(x, y, geotransform, bands):
    col = int((x - geotransform[0]) / geotransform[1])
    row = int((y - geotransform[3]) / geotransform[5])
    if row < 0 or row >= bands[0].shape[0] or \
        col < 0 or col >= bands[0].shape[1]:
        print(f"Invalid coordinates: x={x}, y={y}, \
            row={row}, col={col}")
        return [0, 0, 0]
    return [int(band[row, col]) for band in bands]
```

4. Next, we define the `colorize_las` function to apply color to an LAS file using a GeoTIFF image. Inside this function, we read the GeoTIFF file, read the LAS file, create a new array to store RGB values, iterate through the points to fetch RGB values, create a new LAS file with the same header, set the RGB values, and write the new LAS file:

```
def colorize_las(las_path, geotiff_path, output_path):
    bands, geotransform = read_geotiff(geotiff_path)
    las_file = laspy.read(las_path)
    scale_x, scale_y = las_file.header.scale[0:2]
    offset_x, offset_y = las_file.header.offset[0:2]
    rgb_values = np.zeros((len(las_file.points), 3), \
```

```
dtype=np.uint16)
for i, point in enumerate(las_file.points):
    x = point.X * scale_x + offset_x
    y = point.Y * scale_y + offset_y
    rgb = get_pixel_value(x, y, geotransform, bands)
    rgb_values[i] = rgb
new_las = laspy.create(file_version=\n    las_file.header.version,\n                           point_format=las_file.header.point_\nformat)
new_las.points = las_file.points
new_las.header = las_file.header
new_las.red = rgb_values[:, 0]
new_las.green = rgb_values[:, 1]
new_las.blue = rgb_values[:, 2]
new_las.write(output_path)
```

5. Finally, we provide an example usage of the `colorize_las` function, demonstrating how to call it with specific paths for the LAS and GeoTIFF files:

```
las_path = "DeerIslandLidar.las"
geotiff_path = "DeerIslandAerial.tif"
output_path = "DeerIslandColorLidar.las"
colorize_las(las_path, geotiff_path, output_path)
```

If we look at the new output LiDAR file named `DeerIslandColorLidar.las`, at first glance it looks almost identical to our aerial image:



Figure 8.19 – The top view of our LiDAR file colorized with an aerial image

But if we look at it from the side profile view, we can see it is a 3D model and the points are the colors of the object that the laser measurement reflected from:



Figure 8.20 – The colorized profile view of the LiDAR data

This technique gives us a nice visual three-dimensional model, but remember in remote sensing, we can go beyond the visual and use these models as data. In the next section, we'll do just that and classify this LiDAR dataset so we can potentially use it in automated computer models.

## Classifying LiDAR

Classifying these point clouds is a crucial step in the analysis process, allowing us to distinguish between different types of objects and features within the scanned environment. Ground, vegetation, buildings, and water bodies can all be identified and categorized based on their geometrical properties and reflection characteristics. In this section, we will use a simple technique for LiDAR classification called threshold-based classification. This is the same classification technique we used to classify a satellite image in *Chapter 7, Python and Remote Sensing*. It is a simple and valid technique; however, classification algorithms used by geospatial professionals are usually far more complex and sophisticated. But this method will give you a clear understanding of the approach.

In the following code, we'll assign our LiDAR points one of three classes: ground, vegetation, or trees. The LAS specification has numerous classes assigned to different numbers. In our case, we're going to use three elevation ranges and assign them to class 2, which is "ground" in the specification, class 4, which is "mid vegetation," which we'll refer to as "vegetation," and class 5, which is "high vegetation," which we'll call "trees":

1. First, we import the required libraries – `laspy` for reading and writing LAS files, and `numpy` for array operations:

```
import laspy  
import numpy as np
```

2. Next, we define the `classify` function, which takes an input file, an output file, and a threshold as parameters. This function will handle the entire classification process:

```
def classify(input_file, output_file, threshold):
```

3. Inside the function, we start by reading the input LAS file using Laspy's `read` method:

```
infile = laspy.read(input_file)
```

4. Then, we extract the Z-values from the LAS file. These values represent the elevation of each point:

```
z_values = infile.z
```

5. Now, we classify the points with Z-values within each threshold provided. We use a loop to iterate through the classes, and for each class, we apply a logical comparison using NumPy's `logical_and` method:

```
for c in classes:  
    lower, upper, classification = c  
    points = np.logical_and(lower < z_values, \  
                           z_values < upper)  
    infile.classification[points] = classification
```

6. Next, we create an output LAS file with the same format and version as the input file, copying the points from the input file to the output file:

```
outfile = laspy.create(point_format=\  
                      infile.point_format, \  
                      file_version=infile.header.version)  
outfile.points = infile.points
```

7. Finally, we write the classified data to the output file and print a completion message:

```
outfile.write(output_file)  
print(f"Classification completed. \  
      Output written to {output_file}")
```

When we look at the previous classified LiDAR data with colors assigned to our three classes, we can see the ground (and low vegetation) are blue, the mid-level vegetation is dark green, and the tree canopies are light green:

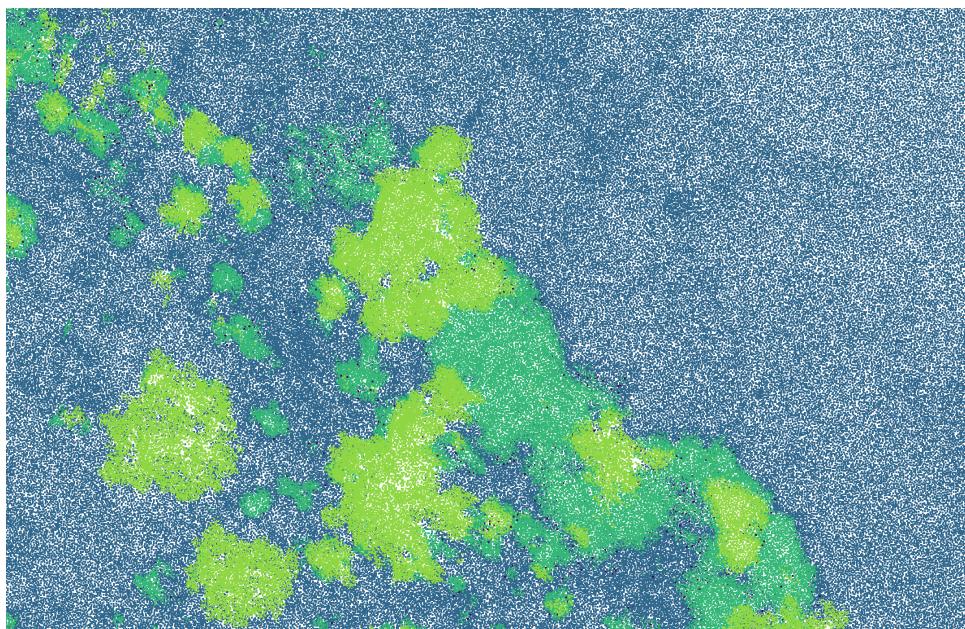


Figure 8.21 – The classified LiDAR image from overhead

We can also look once again at the LiDAR data from a profile view and clearly see the separation of the classes, which correspond to significant features on the island:

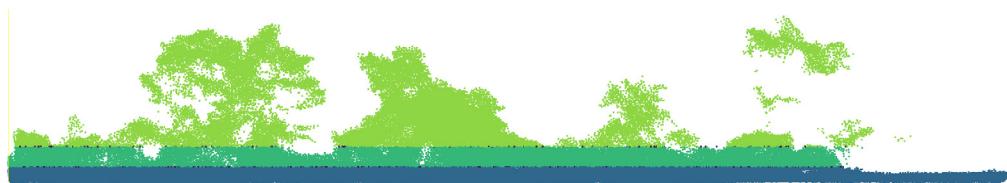


Figure 8.22 – Profile of the classified LiDAR

So far in this chapter, our elevation data work has been on land. But all of these techniques can also be used underwater for the seafloor! In the next section, we'll try our hand at mapping the bottom of the ocean.

## Working with bathymetry

Bathymetry data and terrestrial elevation data share some fundamental similarities, both representing the topography of a surface. While terrestrial data maps the contours of Earth's land surfaces, bathymetry data plunges beneath the waves to depict the underwater terrain, capturing the depth and shapes of ocean floors, lake beds, and river channels. Both types of data are critical for various applications,

from navigation and flood modeling to environmental studies and urban planning. However, the differences in the mediums they represent—air for terrestrial and water for bathymetry—impose unique challenges in data collection and interpretation.

As we've seen, terrestrial data is often collected using LiDAR technology, which employs lasers to send out pulses of light that bounce off the ground and other objects. By measuring the time it takes for the light to return, the distance to the surface can be calculated, resulting in a highly detailed point cloud that represents the land's surface contours.

In contrast, bathymetry data is gathered using **sonar** (short for **sound navigation and ranging**) systems, which use sound waves instead of light. These sound waves travel through the water, reflecting off the underwater terrain. By analyzing the time it takes for the sound waves to return, the depth and shape of the underwater surface can be determined.

Despite the differences in the mediums and energy types they use—light for terrestrial scans and sound for bathymetry—both LiDAR and sonar techniques generate point clouds. These point clouds are versatile and rich datasets that can be further processed and analyzed to create detailed 3D models or rasterized surfaces, bridging the realms of land and water in geospatial analysis.

In this section, we will explore a Python coding example that translates bathymetry data from an `xyz` file into an ASCII Grid. This conversion brings bathymetry data into a familiar format, allowing us to apply the same techniques we've learned for handling terrestrial elevation data. By representing underwater terrains in grid form, we can leverage established GIS techniques for analysis and visualization, creating a bridge between the domains of land and water. The code we'll review doesn't just showcase a practical tool for working with bathymetry data; it underscores the underlying unity and diversity of geospatial analysis, regardless of whether we're charting mountains or mapping the seabed.

We're including a bathymetry example here because hydrography, the mapping of the seafloor, is becoming a critical endeavor for humankind as we become more reliant on the sea for energy, food, transportation, and commerce while attempting to pursue these activities sustainably.

You can download the `bathymetry.xyz` file from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_08\\_Asset\\_Files/bathymetry.xyz](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_08_Asset_Files/bathymetry.xyz).

In the following code, we'll take a `bathymetry.xyz` file and convert it into an ASCII Grid file on which we will then perform all the other techniques used in this chapter. One important difference of note with this type of data is that bathymetry data is usually represented as negative numbers from a chosen water surface datum while terrestrial elevation data is usually measured in positive numbers above a sea surface datum:

1. First, we import the NumPy library, allowing us to efficiently handle numerical operations:

```
import numpy as np
```

2. Next, we read the `bathymetry.xyz` file, ignoring the fourth field, and store the *x*, *y*, and *z* coordinates in the `xyz_data` variable:

```
xyz_data = np.loadtxt('bathymetry.xyz', usecols=(0, 1, 2))
```

3. Then, we define the resolution of the grid, determining the size of each grid cell:

```
resolution = 1.0
```

4. After that, we calculate the minimum and maximum *x* and *y* coordinates, defining the extent of our grid:

```
x_min, y_min = xyz_data[:, 0].min(), xyz_data[:, 1].min()
x_max, y_max = xyz_data[:, 0].max(), xyz_data[:, 1].max()
```

5. Next, we create bins for the *x* and *y* coordinates using the determined extents and resolution:

```
x_bins = np.arange(x_min, x_max + resolution, resolution)
y_bins = np.arange(y_min, y_max + resolution, resolution)
```

6. Now, we use the `np.histogram2d` function to create a 2D histogram representing the grid, weighting the bins by the Z-values:

```
grid, _, _ = np.histogram2d(xyz_data[:, 1], \
    xyz_data[:, 0], bins=(y_bins, x_bins), \
    weights=xyz_data[:, 2])
count, _, _ = np.histogram2d(xyz_data[:, 1], \
    xyz_data[:, 0], bins=(y_bins, x_bins))
```

7. Then, we divide the grid by the count, but only where the count is not 0, to calculate the average depth for each grid cell:

```
grid[count != 0] /= count[count != 0]
```

8. Next, we replace NaN values and cells where the count was 0 with a nodata value, ensuring that our final grid accounts for areas with no data:

```
grid[np.isnan(grid) | (count == 0)] = -9999
```

9. Before writing to the ASCIIGrid file, we flip the grid along the *y* axis to correct the orientation, ensuring that the data is not upside down:

```
grid = np.flipud(grid)
```

10. Finally, we write the grid to an ASCIIGrid file, including the necessary header information, and save it as `bathymetry.asc`:

```
nrows, ncols = grid.shape
header = f"ncols {ncols}\nnrows {nrows}\nllxcorner {x_min}\n"
nyllcorner {y_min}\ncellsize {resolution}\nNODATA_value -9999\n"
with open('bathymetry.asc', 'w') as file:
    file.write(header)
    np.savetxt(file, grid, fmt="%f")
```

By following these steps, we've transformed the bathymetry XYZ data into an ASCII Grid format with the correct orientation, making it ready for further analysis with various GIS tools. We could now apply all of the examples in this chapter. You are encouraged to give it a shot and see what happens. The following image is the result of combining the hillshade, depth or height-based colorization, and contours in this ASCII Grid, which was then overlayed on a satellite image for reference:



Figure 8.23 – The bathymetry data with hillshade, depth coloring, and contours applied

You now have the tools to process elevation data from the tip of Mount Everest to the bottom of the Mariana Trench in the Pacific Ocean!

## Summary

Elevation data can often provide a complete dataset for analysis and derivative products without any other data. In this chapter, you learned how to read and write ASCII Grids using only NumPy. You also learned how to create shaded reliefs, slope grids, and aspect grids. We created elevation contours using a little-known feature called contour in the GDAL library, which is available for Python.

Next, we transformed LiDAR data into an easy-to-manipulate ASCII Grid. We experimented with different ways to visualize the LiDAR data with PIL. Finally, we created a 3D surface or TIN by turning a LiDAR point cloud into a 3D shapefile of polygons. Then we colorized LiDAR using aerial images to create an almost photo-realistic 3D model. We also classified LiDAR so it can be an input to more sophisticated analysis models. And we saw that ocean seafloor data can be processed in much the same way as terrestrial data. These are the tools of terrain analysis that are used for transportation planning, construction planning, hydrological drainage modeling, geologic exploration, ocean mapping, and more.

In the next chapter, we'll combine the building blocks from the previous three chapters to perform some advanced modeling and actually create some information products.

## Questions

1. What is the easy-to-use human- and machine-readable ASCII file format for elevation grid data?
2. What are the two intermediate elevation analysis types we must first calculate when creating a shaded relief from elevation data?
3. What do we call a vector representation of a point dataset in a vector surface of points connected as triangles that allows us to do vector-based analysis on elevation data?
4. What is elevation data collected along the bottom of the ocean called?

## Further reading

- You can find some additional tutorials on Python and elevation data at the following link: <https://www.earthdatascience.org/tutorials/python/elevation/>
- There is a very powerful program with Python integration called PDAL, which is used for creating LiDAR processing workflows and has good tutorials on its site: <https://pdal.io/en/latest/>
- The NOAA **National Centers for Environmental Information (NCEI)** has an excellent online bathymetry viewer called **The Bathymetric Data Viewer** where you can explore and download bathymetric data collected by NOAA: <https://www.ncei.noaa.gov/maps/bathymetry>

# Part 3:

## Practical Geospatial Processing Techniques

This part is of an advanced level, and it will require all of the skills you learned previously. It starts off by teaching you how to create geospatial models to answer specific questions. Furthermore, it will show you a few techniques to build geospatial models and how they will help to predict the future, using visualization concepts. Then, we'll move on to accessing and processing real-time data. At the end of this part, we'll combine all that we learned in the previous sections and implement a system to create an outdoor running or hiking report, based on GPS data and geotagged photos.

This part contains the following chapters:

- *Chapter 9, Advanced Geospatial Modeling*
- *Chapter 10, Working with Real-Time Data*
- *Chapter 11, Putting It All Together*



# 9

## Advanced Geospatial Modeling

In this chapter, we'll build on the data processing concepts that we learned about in the previous chapter in order to create some full-scale information products. The previously introduced data processing methods rarely provide answers to questions by themselves. You can combine these data processing methods to build a geospatial model from multiple processed datasets. A geospatial model is a simplified representation of some aspect of the real world that helps us answer one or more questions about a project or problem. In this chapter, we will introduce some important geospatial algorithms that are commonly used in agriculture, emergency management, logistics, and other industries.

The topics that we will cover are as follows:

- Creating a normalized difference vegetation index (NDVI)
- Creating a flood inundation model
- Creating a color hillshade
- Performing least cost path analysis
- Converting the least cost path to a shapefile
- Routing along streets
- Geolocating photos
- Calculating satellite image cloud cover

The examples in this chapter are longer and more engaging than in the previous chapters. For that reason, there are far more code comments to make the programs easier to follow. We will also use more functions in these examples. In previous chapters, functions were mostly avoided for clarity, but these examples are sufficiently complex, so certain functions make the code easier to read. These examples are actual processes that you would use on the job as a geospatial analyst.

## Technical requirements

In this chapter, we will use the `rasterio` module, which we introduced in *Chapter 4, Geospatial Python Toolbox*. However, if you've installed it using Conda, it may be several versions behind the latest version that we'll need for this chapter. Regardless of how you installed it, you can ensure you have the latest version by running the following:

```
pip install rasterio -U
```

That command will force `pip` to upgrade to the latest version from PyPy if your version is older. At that point, you will only be able to manage the package using `pip`.

The code files for this chapter are available on GitHub: [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730\\_09\\_Asset\\_Files](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730_09_Asset_Files).

## Creating a normalized difference vegetation index (NDVI)

Our first example will be a **normalized difference vegetation index (NDVI)**. NDVIs are used to show the relative health of plants in an area of interest. An NDVI algorithm uses satellite or aerial imagery to show relative health by highlighting the chlorophyll density in plants. NDVIs use only the red and near-infrared bands. The formula of an NDVI is as follows, which we'll refer to as *Eq (1)*:

$$\text{NDVI} = (\text{Infrared} - \text{Red}) / (\text{Infrared} + \text{Red})$$

The goal of this analysis is to produce, to begin with, a multispectral image containing infrared and red bands, and end up with a pseudo color image using seven classes, which color the healthier plants darker green, less-healthy plants lighter green, and bare soil brown.

Because the health index is relative, it is important to localize the area of interest. You could perform a relative index for the entire globe, but vast areas such as the Sahara desert, on the low-vegetation extreme, and densely forested areas such as the Amazon jungle, would skew the results for vegetation in the middle range. However, that being said, climate scientists routinely create global NDVIs to study worldwide trends. The more common application, though, is for managed areas, such as a forest or a farming field, as in this example.

We will begin with an analysis of a single farming field in the Mississippi Delta. To do so, we'll start with a multispectral image of a fairly large area and use a shapefile in order to isolate a single field. The image in the following screenshot is our broad area, with the field of interest highlighted in yellow:

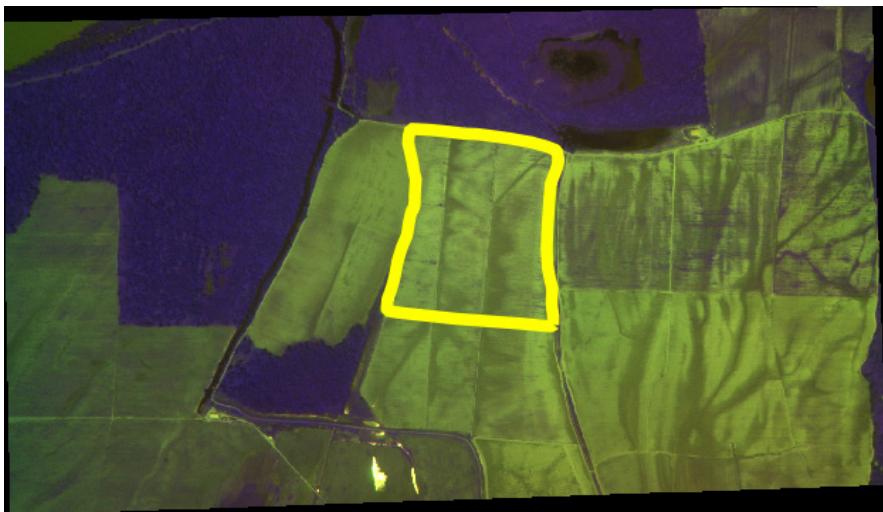


Figure 9.1 – A section of farmland with the highlighted field that we will process

You can download this image and the shapefile for the farming field as a ZIP file from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_09\\_Asset\\_Files/NDVI.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_09_Asset_Files/NDVI.zip)

For this example, we'll use GDAL, OGR, `gdal_array/numpy`, and the **Python Imaging Library (PIL)** to clip and process the data. In the other examples in this chapter, we'll just use simple ASCII grids and NumPy. As we'll be using ASCII elevation grids, GDAL isn't required. In all examples, the scripts use the following convention:

1. Import libraries.
2. Define functions.
3. Define global variables, such as filenames.
4. Execute the analysis.
5. Save the output.

Our approach to the crop health example is split into two scripts. The first script creates the index image, which is a grayscale image. The second script classifies the index and outputs a colored image. In this first script, we'll execute the following steps to create the index image:

1. Read the infrared band.
2. Read the field boundary shapefile.
3. Rasterize the shapefile to an image.
4. Convert the shapefile image to a NumPy array.

5. Use the NumPy array to clip the red band to the field.
6. Do the same for the infrared band.
7. Use the band arrays to execute the NDVI algorithm in NumPy.
8. Save the resulting indexing algorithm to a GeoTIFF file using `gdal_array`.

We will discuss this script in sections to make it easier to follow.

## Setting up the framework

Setting up the framework will help us to import the modules that we need and set up the functions that we'll use for *steps 1* to *5* of the preceding instructions. The `imageToArray()` function converts a PIL image to a NumPy array and is dependent on the `gdal_array` and PIL modules. The `world2Pixel()` function converts geospatial coordinates to the pixel coordinates of our target image. This function uses the georeferencing information that is presented by the `gdal` module. The `copy_geo()` function copies the georeferencing information from our source image to our target array but accounts for the offset that is created when we clip the image. These functions are fairly generic and can serve a role in a variety of different remote sensing processes beyond this example:

1. First, we import the necessary modules. These include GDAL for geospatial operations and NumPy for numerical calculations:

```
from osgeo import gdal_array as gd
from osgeo import gdal
import operator
from osgeo import gdal, gdal_array, ogr
from PIL import Image, ImageDraw
import numpy as np
```

2. Next, we configure GDAL for better error handling by silencing its warnings:

```
gdal.PushErrorHandler('CPLQuietErrorHandler')
gdal.UseExceptions()
```

3. Then, we define a function, `imageToArray`, that converts an image to a NumPy array. This is helpful for image manipulation:

```
def imageToArray(i):
    a = gdal_array.numpy.frombuffer(i.tobytes(), 'b')
    a.shape = i.size[1], i.size[0]
    return a
```

4. Now, let's define another function, `world2Pixel`, that takes in geographical coordinates and a transformation matrix and returns pixel coordinates:

```
def world2Pixel(geoMatrix, x, y):  
    ulX = geoMatrix[0]  
    ulY = geoMatrix[3]  
    xDist = geoMatrix[1]  
    yDist = geoMatrix[5]  
    pixel = int((x - ulX) / xDist)  
    line = int((ulY - y) / abs(yDist))  
    return (pixel, line)
```

5. We also have a function called `copy_geo`, which copies geotransform information from a prototype dataset:

```
def copy_geo(array, prototype=None, xoffset=0, yoffset=0):  
    ds = gdal_array.OpenArray(array)  
    prototype = gdal.Open(prototype)  
    gdal_array.CopyDatasetInfo(prototype, ds, \  
        xoff=xoffset, yoff=yoffset)  
    return ds
```

The next step is to load the data, which we'll be checking in the next section.

## Loading the data

In this section, we load the source image of a farm field using `gdal_array`, which takes it straight into a NumPy array. We also define the name of our output image, which will be `ndvi.tif`. One interesting part of this section is that we load the source image a second time using the GDAL module, as opposed to `gdal_array`.

This second call is to capture the georeferencing data for the image that is available through GDAL, and not `gdal_array`. Fortunately, GDAL only loads raster data on demand, so this approach avoids loading the complete dataset into the memory twice. Once we have the data as a multidimensional NumPy array, we split out the red and infrared bands, as they will both be used in the NDVI equation:

1. We specify the input and output image filenames. The input image must have both red and infrared bands:

```
source = "farm.tif"  
target = "ndvi.tif"
```

2. We then load the source image as a NumPy array and also as a GDAL object to fetch its geotransform data:

```
srcArray = gdal_array.LoadFile(source)
srcImage = gdal.Open(source)
geoTrans = srcImage.GetGeoTransform()
```

3. Now, we extract the red and infrared bands from the source array:

```
r = srcArray[1]
ir = srcArray[2]
```

Now that we have our data loaded, we can turn our shapefile into a raster.

## Rasterizing the shapefile

This section begins the process of clipping. However, the first step is to rasterize the shapefile that outlines the boundary of the specific area that we are going to analyze. That area is within the larger `field.tif` satellite image. In other words, we convert it from vector data to raster data. But we also want to fill in the polygon when we convert it so that it can be used as an image mask. The pixels in the mask will be correlated to the pixels in the red and infrared arrays. Any pixels outside the mask will be turned into nodata pixels so they are not processed as part of the NDVI. To make this correlation, we'll need the solid polygon to be a NumPy array, just like the raster bands. This approach will make sure our NDVI calculation will be limited to the farming field.

The easiest way to convert the shapefile polygon into a filled polygon as a NumPy array is to plot it as a polygon in a PIL image, fill that polygon in, and then convert it to a NumPy array using existing methods, in both PIL and NumPy, that allow that conversion.

In this example, we use the `ogr` module to read the shapefile, because we already have GDAL available. But, we could have also used `PyShp` to read the shapefile just as easily. If our farming field image was available as an ASCII grid, we could have avoided using the GDAL, `gdal_array`, and `ogr` modules altogether:

1. Let's open the shapefile that defines the field boundary and convert its extent to pixel coordinates:

```
field = ogr.Open("field.shp")
lyr = field.GetLayer("field")
poly = lyr.GetNextFeature()
minX, maxX, minY, maxY = lyr.GetExtent()
ulX, ulY = world2Pixel(geoTrans, minX, maxY)
lrX, lrY = world2Pixel(geoTrans, maxX, minY)
```

2. Afterward, we prepare an empty image array where we will store the clipped red and infrared bands:

```
pxWidth = int(lrX - ulX)
pxHeight = int(lrY - ulY)
```

```
clipped = gdal_array.numpy.zeros((3, pxHeight, pxWidth), \
gdal_array.numpy.uint8)
```

3. Now, let's clip the red and infrared bands and update the geotransform information:

```
rClip = r[uly:lrY, ulX:lrX]
irClip = ir[uly:lrY, ulX:lrX]
geoTrans = list(geoTrans)
geoTrans[0] = minX
geoTrans[3] = maxY
```

4. We proceed to map the points to pixels, and then draw the field boundary on a blank image to create a mask:

```
points = []
pixels = []
geom = poly.GetGeometryRef()
pts = geom.GetGeometryRef(0)
for p in range(pts.GetPointCount()):
    points.append((pts.GetX(p), pts.GetY(p)))
for p in points:
    pixels.append(world2Pixel(geoTrans, p[0], p[1]))
rasterPoly = Image.new("L", (pxWidth, pxHeight), 1)
rasterize = ImageDraw.Draw(rasterPoly)
rasterize.polygon(pixels, 0)
mask = imageToArray(rasterPoly)
```

Now that we have converted the shapefile to a mask image, we can clip the bands.

## Clipping the bands

We will clip the bands to the boundary of the mask. For this process, we use NumPy's `choose()` method, which correlates the mask cell to the raster band cell and returns that value or returns 0. The result is a new array that is clipped to the mask, but with the correlated values from the raster band:

```
rClip = np.choose(mask, (rClip, 0)).astype(float)
irClip = np.choose(mask, (irClip, 0)).astype(float)
```

We now have just the data that we want, so we can apply our NDVI relative vegetation health formula.

## Using the NDVI formula

Our final process for creating the NDVI is to execute the NDVI equation (*Eq(1)*). We'll save the output as `ndvi.tif`, and that will be the input for the next script in order to classify and colorize the NDVI as follows:

1. We calculate the NDVI using the formula in *Eq (1)*. In the NDVI equation, both the numerator and denominator could potentially be zero, especially in areas where there is no vegetation or in masked regions. Dividing by zero would result in undefined or infinite values, which would corrupt the NDVI calculation. To address this possibility, we add a small constant (epsilon) to the denominator. The value of epsilon is set to 1 times  $10^{-5}$ , which is a small enough number to not significantly affect the calculation but large enough to prevent division by zero. This practice is a common numerical stability technique. By adding epsilon, we ensure that our program doesn't run into errors or warnings related to division by zero, thereby making our NDVI calculation more robust:

```
epsilon = 1e-5
ndvi = (irClip - rClip) / (irClip + rClip + epsilon).
astype(gdal_array.numpy.float32)
```

2. To finish, we save the NDVI as a GeoTIFF file while maintaining the georeferencing information from the source image:

```
gtiff = gdal.GetDriverByName('GTiff')
gtiff.CreateCopy(target, copy_geo(ndvi, prototype=source,
xoffset=ulX, yoffset=ulY))
gtiff = None
```

The following figure is the output of this example. You need to view it in a geospatial viewer such as QGIS. The image won't open in most image editors. The lighter the shade of gray, the healthier the plants within that field:

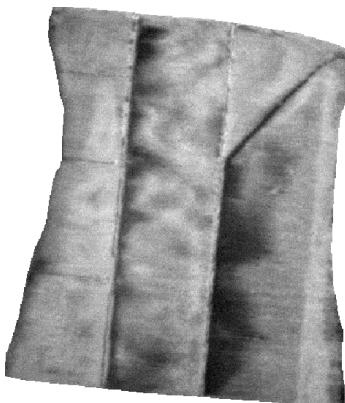


Figure 9.2 – The grayscale NDVI image that we will colorize to make interpretation easier

This image is a good start, but there's more we can do to make it a better visualization.

## Classifying the NDVI

We now have a valid index, but it is not easy to understand, because it is a grayscale image. If we color the image in an intuitive way, then even a child could identify the healthier plants. In the following section, *Additional functions*, we will read in this grayscale index and classify it from brown to dark green using seven classes. The classification and image processing routines, such as the histogram and stretching functions, are almost identical to what we used in the *Creating image histograms* section in *Chapter 7, Python and Remote Sensing*, but this time we are applying them in a much more specific way.

The output of this example will be another GeoTIFF file, but this time it will be a colorful RGB image.

### ***Additional functions***

We won't need any of the functions from our previous NDVI script, but we do need to add a function for creating and stretching a histogram. Both of these functions work with NumPy arrays. We'll also shorten the reference to `gdal_array` to `gd` in this script because it is a long name, and we need it throughout the script:

1. First, we import the required libraries. This includes the GDAL library for geospatial operations and Python's `functools` and `operator` for functional and arithmetic operations:

```
from osgeo import gdal_array as gd
from osgeo import gdal
import operator
from functools import reduce
```

2. Next, we configure GDAL for better error handling:

```
gdal.UseExceptions()
```

3. Then, we define a function named `histogram` to compute the histogram of a multi-dimensional array:

```
def histogram(a, bins=list(range(256))) :
    fa = a.flat
    n = gd.numpy.searchsorted(gd.numpy.sort(fa), bins)
    n = gd.numpy.concatenate([n, [len(fa)]])
    hist = n[1:]-n[:-1]
    return hist
```

4. Now, we have another function named `stretch` that performs histogram stretching on an array. This enhances the contrast in our image:

```
def stretch(a) :
    hist = histogram(a)
```

```

    lut = []
    for b in range(0, len(hist), 256):
        step = reduce(operator.add, hist[b:b+256]) / 255
        n = 0
        for i in range(256):
            lut.append(n / step)
            n = n + hist[i+b]
    gd.numpy.take(lut, a, out=a)
    return a

```

### ***Loading the NDVI***

Next, we'll load the output of our NDVI script back into a NumPy array. We'll also define the name of our output image as `ndvi_color.tif`, and create a zero-filled multidimensional array as a placeholder for the red, green, and blue bands of the colorized NDVI image:

1. The following code will load the NDVI TIFF image into a NumPy array:

```

source = "ndvi.tif"
target = "ndvi_color.tif"

```

2. We then load the NDVI image and convert its data type to `float32`:

```
ndvi = gd.LoadFile(source).astype(gd.numpy.float32)
```

Now that our image is loaded as an array, we can stretch it.

### ***Preparing the NDVI***

We need to perform several steps on the NDVI to prepare it for color visualization, including normalizing it and stretching it:

1. We normalize the NDVI values to be between 0 and 255, but only for the non-zero values, so all the values are within the range of RGB color values:

```

ndvi_normalized = gd.numpy.zeros(ndvi.shape, dtype=gd.numpy.uint8)
mask = ndvi != 0.0
ndvi_normalized[mask] = ((ndvi[mask] + 1) / 2.0 * \
255).astype(gd.numpy.uint8)

```

2. Now, we apply histogram stretching to the normalized NDVI to improve its visual quality and ensure that the image covers the range of classes that will give the final product meaning:

```
ndvi_normalized = stretch(ndvi_normalized)
```

3. We create a three-band image filled with zeros that has the same dimensions as the NDVI image. This will be our RGB output:

```
rgb = gd.numpy.zeros((3, len(ndvi), len(ndvi[0])), \
gd.numpy.uint8)
```

4. We set NDVI values of zero to black in the RGB output, which will make it easier to make them transparent in a GIS map:

```
mask_zero = ndvi_normalized == 0
for j in range(3):
    rgb[j] = gd.numpy.where(mask_zero, 0, rgb[j])
```

### ***Creating classes***

Now we'll set up the structure of image values and colors we will use to classify the image:

1. We define the seven classes and their corresponding RGB colors:

```
classes = [58, 73, 110, 147, 184, 220, 255]
lut = [[120, 69, 25], [255, 178, 74], \
[255, 237, 166], [173, 232, 94], \
[135, 181, 64], [3, 156, 0], [1, 100, 0]]
```

2. We initialize a variable, `start`, to define the starting value for class ranges:

```
start = 1
```

### ***Classifying the image***

We've loaded and prepared our data, and we've set up our classes, so now we can do what we set out to do and classify the image:

1. First, we classify each NDVI range based on our predefined lookup table:

```
for i in range(len(classes)):
    mask = gd.numpy.logical_and\
        (start <= ndvi_normalized, \
        ndvi_normalized <= classes[i])
```

2. Now we can colorize the classes:

```
for j in range(3):
    rgb[j] = gd.numpy.where(mask, lut[i][j], rgb[j])
    start = classes[i] + 1
```

### Saving the output

Finally, we save the classified and colorized image as a GeoTIFF:

```
gd.SaveArray(rgb, target, format="GTiff", prototype=source)
```

Here is the image that we output, with green showing healthier vegetation, yellow showing less healthy vegetation, and brown showing bare soil:



Figure 9.3 – The classified NDVI

This is our final product for this example. Farmers can use this data to determine how to effectively irrigate and spray chemicals, such as fertilizers and pesticides, in a targeted, more effective, and more environment-friendly way. In fact, these classes can even be turned into a vector shapefile, which is then loaded into a GPS-driven computer on a field sprayer. This automatically applies the correct amount of chemicals in the correct places as a sprayer is driven around the field, or in some cases, even flown over the field in an airplane with a sprayer attachment.

Notice as well that even though we clipped the data to the field, the image is still a square. The black areas are the nodata values that have been converted to black. In display software, you can make the nodata color transparent without affecting the rest of the image.

Although we created a very specific type of product, a classified NDVI, the framework of this script can be altered in order to implement many remote sensing analysis algorithms. There are different types of NDVIs, but with relatively minor changes, you can turn this script into a tool that can be used to look for harmful algae blooms in the ocean or smoke in the middle of a forest indicating a forest fire.

### Clipping with GDAL

This book attempts to limit the use of GDAL as much as possible in order to focus on what can be accomplished with pure Python and tools that can easily be installed from PyPI. However, it is helpful to remember that there is a wealth of information on using GDAL and its associated utilities to carry out similar tasks. For another tutorial on clipping a raster with GDAL via its command-line utilities, see [https://joeyklee.github.io/broc-cli-geo/guide/XX\\_raster\\_cropping\\_and\\_clipping.html](https://joeyklee.github.io/broc-cli-geo/guide/XX_raster_cropping_and_clipping.html).

Now that we've worked with the land, let's work with water in order to create a flood inundation model.

## Creating a flood inundation model

In this next example, we'll begin to enter the world of hydrology. Flooding is one of the most common and devastating natural disasters, which affects nearly every population around the globe. Geospatial models are a powerful tool in estimating the impact of a flood and mitigating that impact before it happens. We often hear on the news that a river is reaching the flood stage, but that information is meaningless if we can't understand the impact.

Hydrological flood models are expensive to develop and can be very complex. These models are essential for engineers in building flood control systems. However, first responders and potential flood victims are only interested in the impact of an impending flood.

We can begin to understand the flooding impact in an area using a very simple and easy-to-comprehend tool called a **flood inundation model**. This model starts with a single point and floods an area with the maximum volume of water that a flood basin can hold at a particular flood stage. Usually, this analysis is a worst-case scenario. Hundreds of other factors go into calculating how much water will enter into a basin from a river-topping flood stage. But we can still learn a lot from this simple first-order model.

### SRTM data

As mentioned in the *Point cloud data* section in *Chapter 1, Learning about Geospatial Analysis with Python*, the **Shuttle Radar Topography Mission (SRTM)** dataset provides a near-global **digital elevation model (DEM)** that you can use for these types of models. More on SRTM data can be found here: <http://www2.jpl.nasa.gov/srtm/>.

You can download the ASCII grid data in EPSG:4326, and a shapefile containing the point as a .zip file from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_09\\_Asset\\_Files/FloodFill.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_09_Asset_Files/FloodFill.zip). The shapefile is just for reference and has no role in this model. The following image is a digital elevation model with a source point displayed as a yellow star near Houston, Texas. In real-world analysis, this point would likely be a stream gauge where you would have data about the river's water level:

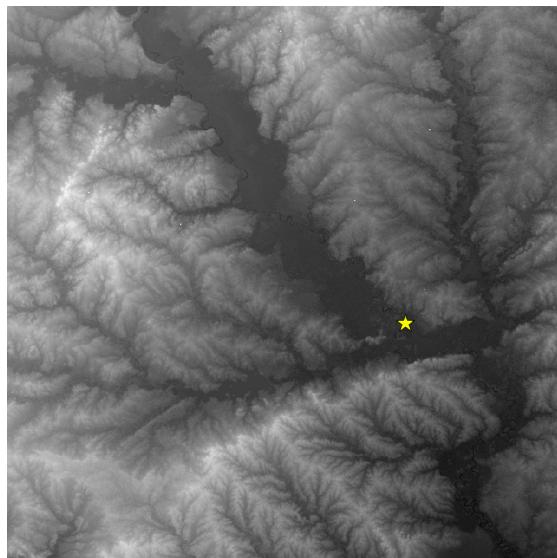


Figure 9.4 – A digital elevation model with a star near Houston, Texas

The algorithm that we are introducing in this example is called a **flood fill algorithm**. This algorithm is well known in the field of computer science and is used in the classic computer game **Minesweeper** to clear empty squares on the board when a user clicks a square. It is also the method that is used for the well-known paint bucket tool in graphics programs such as **Adobe Photoshop**, and it is used to fill an area of adjacent pixels of the same color with a different color.

There are many ways to implement this algorithm. One of the oldest and most common ways is to recursively crawl through each pixel of the image. The problem with recursion is that you end up processing pixels more than once and creating an unnecessary amount of work. The resource usage for a recursive flood fill can easily crash a program on even a moderately sized image.

This script uses a four-way queue-based flood fill that may visit a cell more than once but ensures that we only process a cell once. The queue only contains unique, unprocessed cells by using Python's built-in set type, which only holds unique values. We use two sets: **fill**, which contains the cells we need to fill, and **filled**, which contains processed cells.

This example executes the following steps:

1. Extract the header information from the ASCII DEM.
2. Open the DEM as a NumPy array.
3. Define our starting point as a row and column in the array.
4. Declare a flood elevation value.
5. Filter the terrain to only the desired elevation value and below.

6. Process the filtered array.
7. Create a 1, 0, 0 array (that is, a binary array) with flooded pixels as 1.
8. Save the flood inundation array as an ASCII grid.

This example can take a minute or two to run on a slower machine; we'll use the print statements throughout the script as a simple way to track progress. Once again, we'll break this script up with explanations, for clarity.

Now that we have our data, we can begin our flood fill function.

## The flood fill function

We use ASCII Grids in this example, which means that the engine for this model is completely in NumPy. We start off by defining the `floodFill` function, which is the heart and soul of this model. This Wikipedia article on flood fill algorithms provides an excellent overview of the different approaches: [http://en.wikipedia.org/wiki/Flood\\_fill](http://en.wikipedia.org/wiki/Flood_fill).

Flood fill algorithms start at a given cell and begin checking the neighboring cells for similarity. The similarity factor might be color or, in our case, elevation. If the neighboring cell is of the same or lower elevation as the current cell, then that cell is marked for checks of its neighbor until the entire grid is checked. NumPy isn't designed to crawl over an array in this way, but it is still efficient in handling multidimensional arrays overall. We step through each cell and check its neighbors to the north, south, east, and west. Any of those cells that can be flooded are added to the filled set, and their neighbors are added to the fill set to be checked by the algorithm.

As mentioned earlier, if you try to add the same value to a set twice, it just ignores the duplicate entry and maintains a unique list. By using sets in an array, we efficiently check a cell only once because the fill set contains unique cells. The following code implements our `floodFill` function:

1. First, we import the required libraries. NumPy is used for numerical operations and `getline` from `linecache` will be used to read specific lines from a file:

```
import numpy as np
from linecache import getline
```

2. Next, we define a function called `floodFill`, which takes the `c` column, row `r`, and a `mask` array as arguments. It will return an array marking all points connected to the starting cell (`c, r`) with 1s:

```
def floodFill(c, r, mask):
    filled = set()
    fill = set()
    fill.add((c, r))
    width = mask.shape[1] - 1
    height = mask.shape[0] - 1
```

```

flood = np.zeros_like(mask, dtype=np.int8)
while fill:
    x, y = fill.pop()
    if y == height or x == width or x < 0 or y < 0:
        continue
    if mask[y][x] == 1:
        flood[y][x] = 1
        filled.add((x, y))
        west = (x-1, y)
        east = (x+1, y)
        north = (x, y-1)
        south = (x, y+1)
        if west not in filled:
            fill.add(west)
        if east not in filled:
            fill.add(east)
        if north not in filled:
            fill.add(north)
        if south not in filled:
            fill.add(south)
return flood

```

3. Then, we specify the source and target filenames for our terrain and flood fill output:

```

source = "terrain.asc"
target = "flood.asc"

```

4. We open the image using NumPy's `loadtxt` method after skipping the first six rows, which are part of the header:

```

print("Opening image...")
img = np.loadtxt(source, skiprows=6)
print("Image opened")

```

5. We create a mask where all pixels with an elevation of less than 70 meters are marked as 1 and the rest as 0:

```

a = np.where(img < 70, 1, 0)
print("Image masked")

```

6. Next, we read the header using `getline` to fetch metadata such as the number of columns, rows, and cell size:

```
hdr = [getline(source, i) for i in range(1, 7)]
values = [float(h.split(" ")[-1].strip()) for h in hdr]
cols, rows, lx, ly, cell, nd = values
xres = cell
yres = cell * -1
```

7. We specify the starting coordinates (`sx`, `sy`) for our flood fill operation:

```
sx = 2582
sy = 2057
```

8. We call our `floodFill` function to start the flooding process:

```
print("Beginning flood fill")
fld = floodFill(sx, sy, a)
print("Finished Flood fill")
```

9. We prepare a header string for our output file:

```
header = ""
for i in range(6):
    header += hdr[i]
```

10. Finally, we save the flood fill output to a file, ensuring to include the header:

```
print("Saving grid")
with open(target, "wb") as f:
    f.write(header.encode('UTF-8'))
    np.savetxt(f, fld, fmt="%li")
print("Done!")
```

The image in the following screenshot shows the flood inundation output over a classified version of the DEM, with lower elevation values in brown, mid-range values in green, and higher values in gray and white:

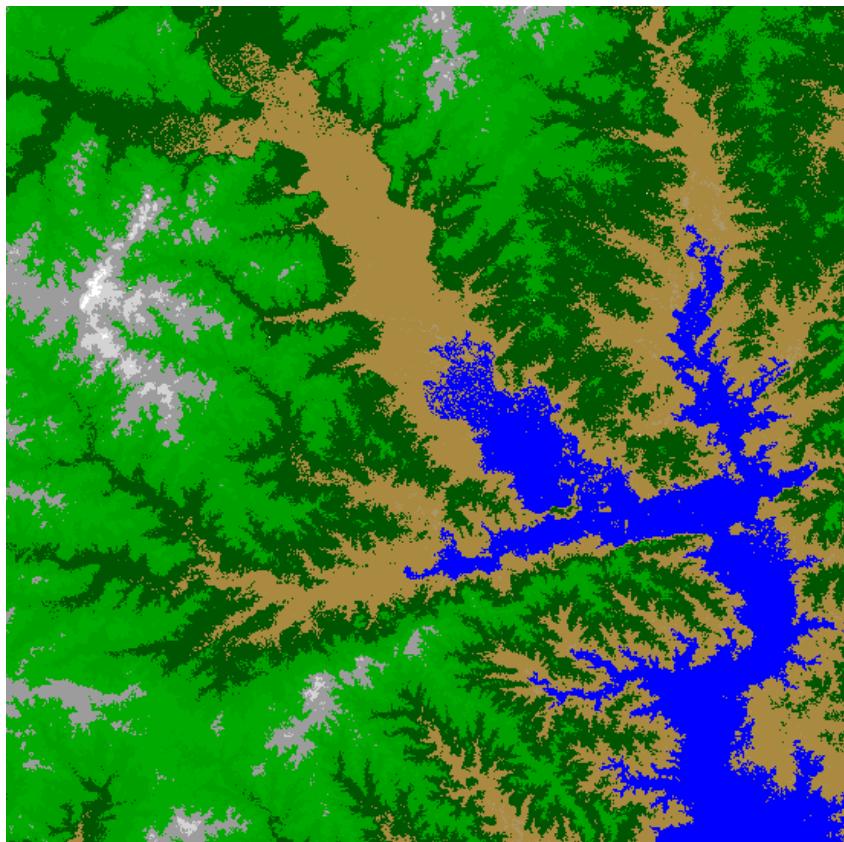


Figure 9.5 – Flood inundation model over a classified DEM

The flood raster, which includes all areas less than 70 meters, is colored blue. This image was created with QGIS, but it could be displayed in ArcGIS as EPSG:4326. You could also use GDAL to save the flood raster grid as an 8-bit TIFF file or JPEG file, just like the NDVI example, in order to view it in a standard graphics program.

This image in the following screenshot is nearly identical, except for the filtered mask from which the inundation was derived, which is displayed in yellow. This is done by generating a file for the array called `wet`, instead of `f1d`, to show the non-contiguous regions, which were not included as part of a flood. These areas are not connected to the source point, so they would unlikely be reached during a flood event unless the event was caused by significant rain in that area and these vulnerable elevations filled up:

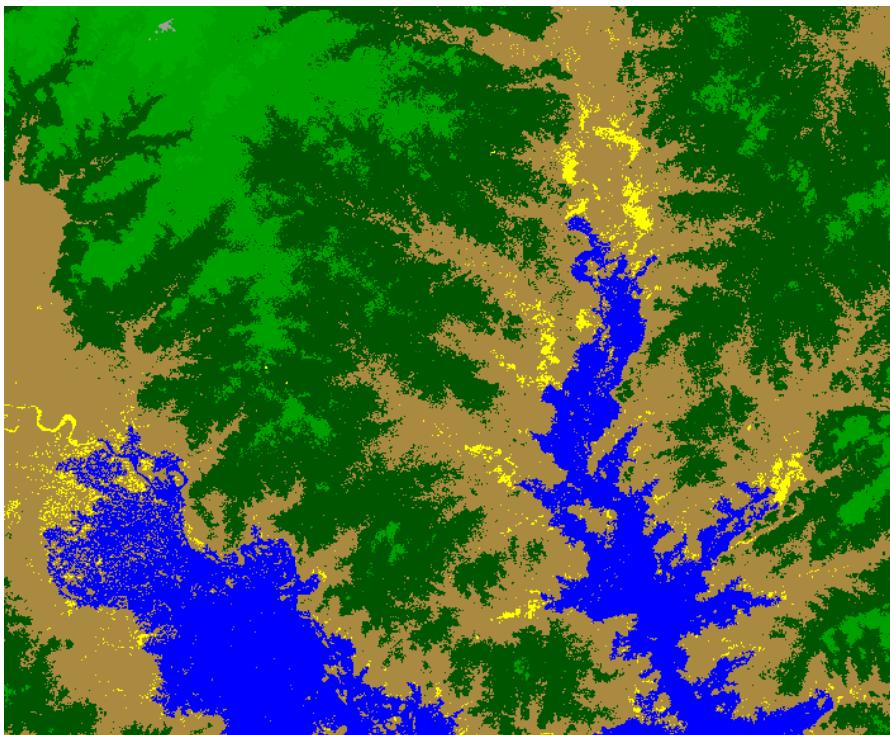


Figure 9.6 – The non-contiguous areas in yellow at the same elevation levels as the flooded area

By changing the elevation value and running the model again, you can create additional flood inundation rasters. We started with an elevation of 70 meters. If we increase that value to 90, we can expand the flood. The following screenshot shows a flood event at both 70 meters in darker blue and 90 meters in lighter blue:

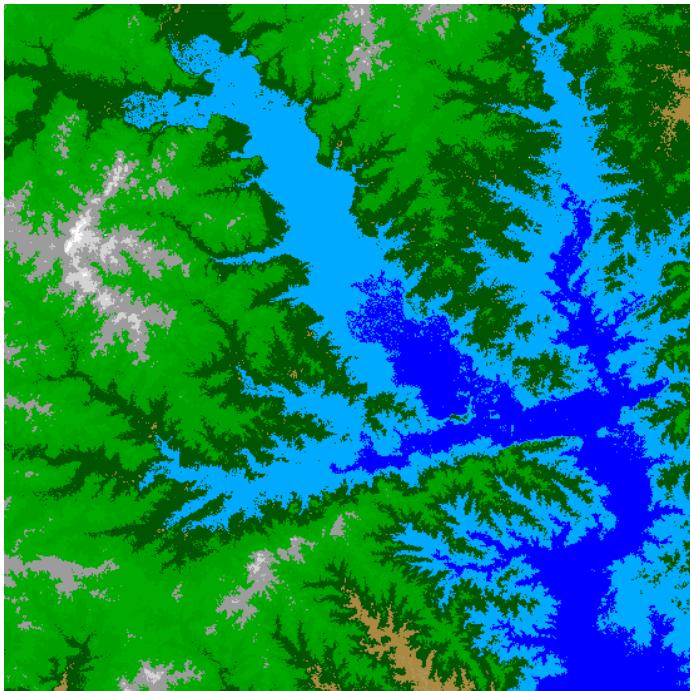


Figure 9.7 – 70-meter flood event (dark blue) and 90-meter flood event (light blue)

This model is an excellent, useful visualization. However, you could take this analysis even further by using GDAL's `polygonize()` method on the flood mask, as we did with the island in the *Extracting features from images* section in *Chapter 7, Python and Remote Sensing*. This operation would give you a vector flood polygon. Then, you could use the principles that we discussed in the *Performing selections* section in *Chapter 6, Creating and Editing GIS Data*, to select buildings using the polygon to determine population impact. You could also combine that flood polygon with the dot density example in *Chapter 6, Creating and Editing GIS Data*, in the *Dot density calculations* section, to assess the potential population impact of a flood. The possibilities are endless!

## Creating a color hillshade

In this example, we'll combine previous techniques to combine our terrain hillshade from *Chapter 8, Python and Elevation Data*, with the color classification that we used on the LIDAR. For this example, we'll need the ASCII Grid DEMs named `dem.asc` and `relief.asc` that we used in the previous chapter.

We'll create a colorized DEM and a hillshade and then use PIL to blend them together for an enhanced elevation visualization. The code comments will guide you through the example, as many of these steps are already familiar to you.

So, let's dive into this code to understand how it generates a hillshade image by blending a DEM with a shaded relief background. The code uses the `PIL` for image manipulation and `GDAL` for geospatial operations:

1. First, we import the necessary libraries. `GDAL` is for geospatial operations, and `PIL` is for image processing:

```
from osgeo import gdal_array as gd
try:
    import Image
except:
    from PIL import Image
```

2. We then specify the filenames for our relief background, DEM, and the resulting hillshade image:

```
relief = "relief.asc"
dem = "dem.asc"
target = "hillshade.tif"
```

3. Next, we load the relief image and the DEM into NumPy arrays, skipping the first six rows, which are header lines:

```
bg = gd.numpy.loadtxt(relief, skiprows=6)
fg = gd.numpy.loadtxt(dem, skiprows=6) [-2:, :-2]
```

4. We create a blank three-band RGB image with the same dimensions as our DEM:

```
rgb = gd.numpy.zeros((3, len(fg), len(fg[0])), gd.numpy.uint8)
```

5. We define a list of classes based on elevation ranges and a corresponding color **look-up table (LUT)**:

```
classes = [356, 649, 942, 1235, 1528, 1821,
2114, 2300, 2700]
lut = [[63, 159, 152], [96, 235, 155], \
[100, 246, 174], [248, 251, 155], [246, 190, 39], \
[242, 155, 39], [165, 84, 26], [236, 119, 83], \
[203, 203, 203]]
```

6. We initialize the starting elevation value for the first class:

```
start = 1
```

7. For each class, we identify the pixels in the DEM that fall within that elevation range and colorize them according to the LUT:

```
for i in range(len(classes)):
    mask = gd.numpy.logical_and(start <= fg, \
```

```
fg <= classes[i])
for j in range(len(lut[i])):
    rgb[j] = gd.numpy.choose(mask, (rgb[j], \
        lut[i][j]))
start = classes[i] + 1
```

8. We then convert the relief and DEM arrays to PIL image objects:

```
im1 = Image.fromarray(bg).convert('RGB')
im2 = Image.fromarray(rgb.transpose(1, 2, 0)).convert('RGB')
We blend the two images together with a 40% alpha to create the
hillshade image.
hillshade = Image.blend(im1, im2, .4)
```

9. Finally, we save the resulting hillshade image as a TIFF file:

```
hillshade.save(target)
```

The following figure shows the output, which makes a great backdrop for GIS maps:



Figure 9.8 – A color hillshade map

Now that we can model terrain, let's learn how to navigate over it.

## Performing least cost path analysis

Calculating driving directions is the most commonly used geospatial function in the world. Typically, these algorithms calculate the shortest path between points A and B, or they may take into account the speed limit of the road, or even current traffic conditions, in order to choose a route by drive time.

But what if your job is to build a new road? Or what if you are in charge of deciding where to run power transmission lines or water lines across a remote area? In a terrain-based setting, the shortest path might cross a difficult mountain, or run through a lake. In this case, we need to account for obstacles and avoid them if possible. However, if avoiding a minor obstacle takes us too far out of our way, the cost of implementing that route may be more expensive than just going over a mountain.

This type of advanced analysis is called **least cost path** analysis. We search an area for the route that is the best compromise of distance versus the cost of following that route. The algorithm that we use for this process is called the **A-star** or **A\* algorithm**. The oldest routing method is called the **Dijkstra algorithm**, which calculates the shortest path in a network, such as a road network. The A\* method can do that as well, but it is also better suited for traversing a grid-like DEM.

### More on routing algorithms

You can find out more about these algorithms on the following web pages:

- Dijkstra's algorithm: [http://en.wikipedia.org/wiki/Dijkstra's\\_algorithm](http://en.wikipedia.org/wiki/Dijkstra's_algorithm)
- A\* algorithm: [http://en.wikipedia.org/wiki/A-star\\_algorithm](http://en.wikipedia.org/wiki/A-star_algorithm)

This example is the most complex in this chapter. To better understand it, we have a simple version of the next program, which is text-based and operates on a 5 x 5 grid with randomly generated values. You can actually see how this program follows the algorithm before trying it on an elevation grid with thousands of values.

This program executes the following steps:

1. Create a simple grid with randomly generated pseudo-elevation values between 1 and 16.
2. Define a start location in the lower-left corner of the grid.
3. Define the endpoint as the upper-right corner of the grid.
4. Create a cost grid that has the elevation of each cell, plus the cell's distance to the finish.
5. Examine each neighboring cell from the start, and choose the one with the lowest cost.
6. Repeat the evaluation using the chosen cell until we get to the end.
7. Return the set of chosen cells as the least cost path.
8. Set up the test grid.

You simply run this program from the command line and view its output. The first section of this script sets up our artificial terrain grid as a randomly generated NumPy array, with notional elevation values between 1 and 16. We also create a distance grid that calculates the distance from each cell to the destination cell. This value is the cost of each cell. Let's have a look at the following steps:

1. First, we import NumPy for numerical operations and heapq for priority queue management:

```
import numpy as np
import heapq
```

2. We define our A\* search function, `astar`, which will take in start and end points, and the heuristic (`h`) and cost (`g`) grids:

```
def astar(start, end, h, g):
```

3. Inside the function, we initialize sets and dictionaries to keep track of the algorithm's state:

```
closed_set = set()
open_set = [(0, start)]
came_from = {}
g_costs = {start: 0}
```

4. The main loop of the algorithm keeps running as long as the open set is not empty:

```
while open_set:
```

5. We pop the cell with the lowest f score from the priority queue:

```
_, current = heapq.heappop(open_set)
```

6. If we've reached the end, we reconstruct the path and return it:

```
if current == end:
    path = []
    while current in came_from:
        path.insert(0, current)
        current = came_from[current]
    return path
```

7. We then identify the neighbors of the current cell, considering boundary conditions:

```
neighbors = []
y, x = current
for dy, dx in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
    neighbor = y + dy, x + dx
    if 0 <= neighbor[0] < h.shape[0] \
    and 0 <= neighbor[1] < h.shape[1]:
        neighbors.append(neighbor)
```

8. Each neighbor is evaluated, and we update the  $f$  score if we find a shorter path to that cell:

```
for neighbor in neighbors:
    if neighbor in closed_set:
        continue
    tentative_g = g_costs[current] + g[neighbor]
    if neighbor not in g_costs \
    or tentative_g < g_costs[neighbor]:
        g_costs[neighbor] = tentative_g
        came_from[neighbor] = current
    f_score = tentative_g + h[neighbor]
    if neighbor not in [i[1] \
    for i in open_set]:
        heapq.heappush(open_set, \
        (f_score, neighbor))
```

9. The current cell is added to the closed set:

```
closed_set.add(current)
```

10. The function will return an empty list if there's no path to the end:

```
return []
```

11. The remainder of the code initializes the grid, cost arrays, and distance heuristic:

```
w = 5
h = 5
start = (h-1, 0)
end = (0, w-1)
a = np.zeros((w, h))
dist = np.zeros(a.shape, dtype=np.int8)
```

12. Here, the program prints the cost grid, which is generated by combining randomly generated values with the distance grid:

```
print("COST GRID (Value + Distance)")
print(cost)
print()
```

13. The A\* algorithm is then called to find the least cost path:

```
path = astar(start, end, cost, dist)
```

14. Finally, we visualize the found path on a grid and print it:

```
print("PATH GRID: 1=path")
print(path_grid)
```

Next, we'll view the output of this test. When you run this program, you'll generate a randomly numbered grid similar to the following:

```
COST GRID (Value + Distance)
[[13 9 10 5 7]
 [13 5 6 15 10]
 [8 16 9 2 15]
 [8 13 5 4 15]
 [1 7 17 5 8]]

PATH GRID: 1=path
[[0 0 1 1 1]
 [0 0 1 0 0]
 [0 0 1 0 0]
 [0 1 1 0 0]
 [0 1 0 0 0]]
```

The grid is small enough that you can easily trace the algorithm's steps manually. This implementation uses **Manhattan distance**, which means the distance does not use diagonal lines—only left, right, up, and down measurements are taken. The search also does not move diagonally, in order to keep things simple. Now that we understand the algorithm, let's try it on real terrain.

## The real-world example

For the real-world example, we'll use the same DEM that is located near Vancouver, British Columbia, Canada, that we used in *Chapter 8, Python and Elevation Data*, in the *Creating a shaded relief* section. The spatial reference for this grid is EPSG:26910 NAD 83/UTM zone 10N. You can download the DEM, relief, and start and end points of the shapefile as a zipped package from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_09\\_Asset\\_Files/dem.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_09_Asset_Files/dem.zip).

We'll actually use the shaded relief for visualization. Our goal in this exercise will be to move from the start to the finish point in the lowest-cost way possible:

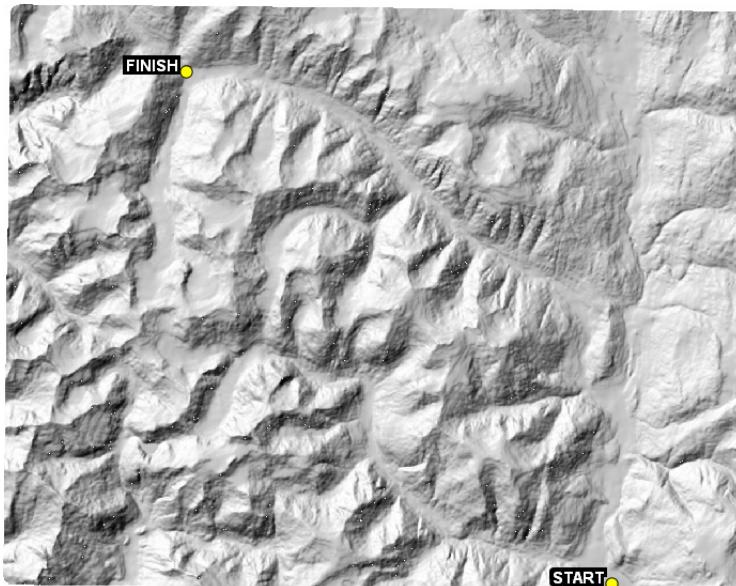


Figure 9.9 – The shaded relief terrain image we will route through

Just looking at the terrain, there are two paths that follow low-elevation routes without much change in direction. These two routes are illustrated in the following screenshot:

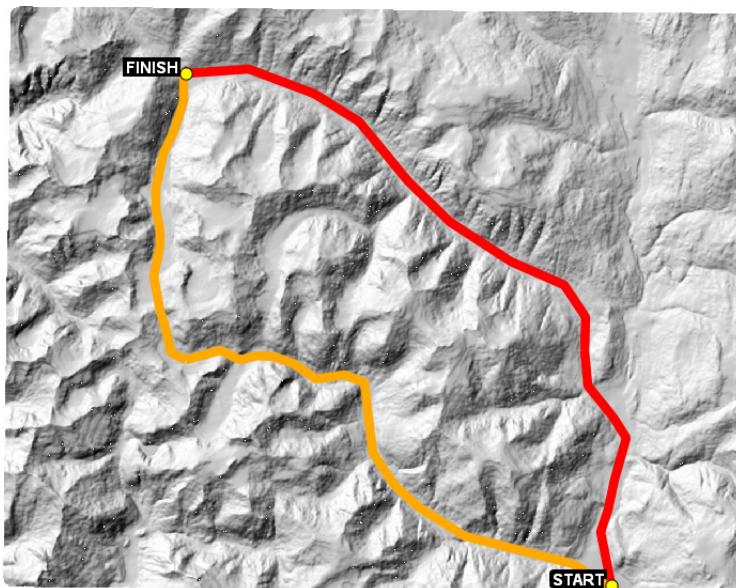


Figure 9.10 – Two obvious possible paths through the terrain

So, we would expect that when we use the A\* algorithm, the result would be close. Remember that the algorithm only looks in the immediate vicinity, so it can't look at the whole image as we can, and it can't make adjustments early in the route based on a known obstacle ahead.

We will expand this implementation from our simple example and use Euclidean distance, or as the crow flies measurements, and we will also allow the search to look in eight directions instead of four. We will prioritize terrain as the primary decision point. We will also use distances, both to the finish and from the start, as lower priorities in order to make sure that we are moving forward toward the goal and not getting too far off track. Other than those differences, the steps are identical to the simple example. The output will be a raster with the path values set to one and the other values set to zero.

Now that we understand the problem, let's solve it!

### ***Loading the grid***

We'll start loading our ASCII grid terrain data using NumPy with the following steps:

1. First, we import necessary libraries such as NumPy for array operations, math for mathematical functions, `linecache` for reading specific lines from a file, and `pickle` for serializing Python objects:

```
import numpy as np
import math
from linecache import getline
import pickle
```

2. Next, we define the source file for the terrain data and the target file for the raster path. We then load the terrain data into a NumPy array, skipping the first six rows as they contain header information:

```
source = "dem.asc"
target = "path.asc"
cost = np.loadtxt(source, skiprows=6)
```

3. Then, we parse the header to obtain metadata such as the number of columns, rows, cell size, and nodata value:

```
hdr = [getline(source, i) for i in range(1, 7)]
values = [float(ln.split(" ")[-1].strip()) for ln in hdr]
cols, rows, lx, ly, cell, nd = values
```

4. Finally, we set the starting and ending positions on the grid:

```
sx = 1006
sy = 954
dx = 303
dy = 109
```

Now that we have our data, let's set up the functions we'll need to process it.

### ***Defining the helper functions***

We'll need a couple of helper functions, as shown next, to calculate the score of each move:

1. We define a function called `e_dist` to calculate the Euclidean distance between two points:

```
def e_dist(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    distance = math.sqrt((x1-x2)**2+(y1-y2)**2)
    return int(distance)
```

2. Next, we define another function called `weighted_score` to calculate the weighted score for each node based on its distance from the start and end points as well as its elevation:

```
def weighted_score(cur, node, h, start, end):
    cur_h = h[cur]
    cur_g = e_dist(cur, end)
    cur_d = e_dist(cur, start)
    node_h = h[node]
    node_g = e_dist(node, end)
    node_d = e_dist(node, start)
    score = 0
    if node_h < cur_h:
        score += cur_h - node_h
    if node_g < cur_g:
        score += 10
    if node_d > cur_d:
        score += 8
    return score
```

We can now implement the A\* algorithm.

### ***The real-world A\* algorithm***

This algorithm is more complex than the simple version in our previous example. We use sets to avoid redundancy. It also implements our more advanced scoring algorithm and checks to make sure we aren't at the end of the path before doing additional calculations. Unlike our last example, this more advanced version also checks cells in eight directions, so the path can move diagonally:

1. We define the `astar` function to implement the A\* algorithm:

```
def astar(start, end, h):
```

2. Inside this function, we initialize sets for the closed set of nodes, the open set of nodes, and the path:

```
closed_set = set()
open_set = set()
path = []
```

3. Then, we add the starting point to the open set and enter a `while` loop that continues until the open set is empty:

```
open_set.add(start)
while open_set:
```

4. Within the loop, we pop a node from the open set and check whether it's the endpoint. If it is, we return the path:

```
cur = open_set.pop()
if cur == end:
    return path
```

5. Next, we add the current node to the closed set and the path:

```
closed_set.add(cur)
path.append(cur)
```

6. Then, we identify the neighbors of the current node and check whether the endpoint is among them. If it is, we return the path:

```
options = []
y1 = cur[0]
x1 = cur[1]
if y1 > 0:
    options.append((y1-1, x1))
if y1 < h.shape[0]-1:
    options.append((y1+1, x1))
if x1 > 0:
    options.append((y1, x1-1))
if x1 < h.shape[1]-1:
    options.append((y1, x1+1))
if x1 > 0 and y1 > 0:
    options.append((y1-1, x1-1))
if y1 < h.shape[0]-1 and x1 < h.shape[1]-1:
    options.append((y1+1, x1+1))
if y1 < h.shape[0]-1 and x1 > 0:
    options.append((y1+1, x1-1))
if y1 > 0 and x1 < h.shape[1]-1:
```

```

        options.append((y1-1, x1+1))
    if end in options:
        return path

```

- We score the neighbors and add the best-scoring neighbor to the open set:

```

best = options[0]
best_score = weighted_score(cur, best, h, \
start, end)
for i in range(1, len(options)):
    option = options[i]
    if option in closed_set:
        continue
    option_score = weighted_score(cur, \
option, h, start, end)
    if option_score > best_score:
        best = option
        best_score = option_score
open_set.add(best)

```

Now we can use the A\* algorithm to create a path.

### ***Generating a real-world path***

Now we'll build the path point by point as we find the least-cost path:

- We call our `astar` function to find the least-cost path and store it in a variable, `p`:

```
p = astar((sy, sx), (dy, dx), cost)
```

- Next, we create a NumPy array to hold our path and set its shape to match our cost grid:

```
path = np.zeros(cost.shape)
```

- Then, we iterate through the nodes in our path and set their corresponding positions in our path grid to 1:

```

for y, x in p:
    path[y][x] = 1
path[dy][dx] = 1

```

- Finally, we save our path grid to a file, preserving the header information from the original terrain data:

```

with open(target, "wb") as f:
    f.write(header, 'UTF-8'))
    np.savetxt(f, path, fmt="%4i")

```

Here is the output route of our search:

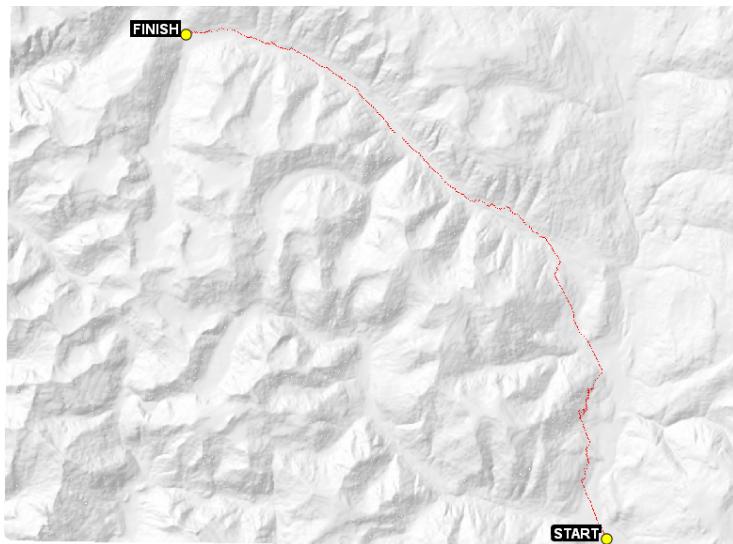


Figure 9.11 – The output of our path search

As you can see, the A\* search came very close to one of our manually selected routes. In a couple of cases, the algorithm chose to tackle some terrain, instead of trying to go around it. Sometimes the slightly hilly terrain is deemed less of a cost than the distance to go around it. You can see examples of that choice in this zoomed-in portion of the upper-right section of the route. The red line is the route that our program generated through the terrain:

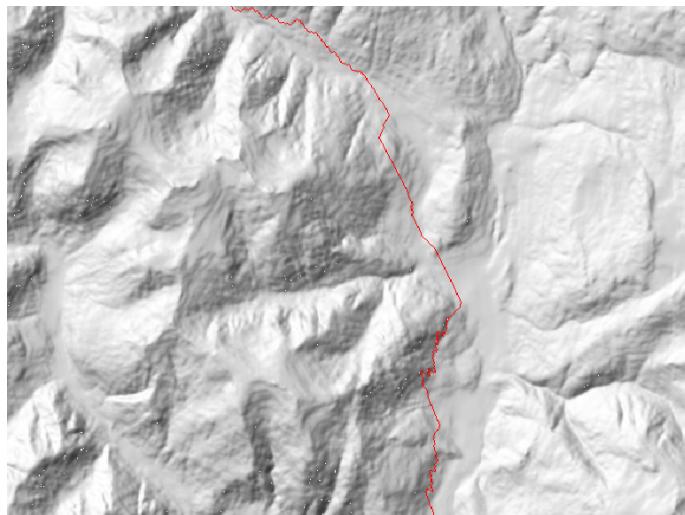


Figure 9.12 – An area where the algorithm decided to climb a slight hill

We only used two values: terrain and distance. But you could also add hundreds of factors, such as soil type, water bodies, and existing roads. All of these items could serve as an impedance or an outright wall. You would just modify the scoring function in the example to account for any additional factors. Keep in mind that the more factors you add, the more difficult it is to trace what the A\* implementation was thinking when it chose the route.

An obvious future direction for this analysis would be to create a vector version of this route as a line. The process would include mapping each cell to a point and then using nearest-neighbor analysis to order the points properly, before saving it as a shapefile or GeoJSON file. So let's give that a shot in the next section.

## Converting the route to a shapefile

The raster version of the least cost path route is useful for visualization, but it isn't very good for analysis because it is embedded in the raster, and it is, therefore, difficult to relate to other datasets as we have done so many other times in this book. Our next goal will be to use the path data that we saved when creating the route to create a shapefile since the saved data is in the proper order. The following code will convert our raster path to a shapefile that is easier to use in a GIS for analysis:

1. First, let's import the necessary modules. We'll need `pickle` to load the previously saved path, `getline` from `linecache` to read the header information from a file, and `shapefile` to write shapefiles:

```
import pickle
from linecache import getline
import shapefile
```

2. Next, we define a function named `pix2coord` to convert pixel coordinates to real-world coordinates. This function takes in three parameters: `gt` for geotransform, `x` for the pixel's `x` coordinate, and `y` for the pixel's `y` coordinate:

```
def pix2coord(gt,x,y):
    w = gt[0]
    h = gt[1]
    ox = gt[2]
    oy = gt[3]
    pw = gt[4]
    ph = gt[5]
    cx = ox + (x * pw)
    cy = (oy + (ph*h)) - (y * ph)
    return cx, cy
```

3. Then, we read the path from a previously saved `pickle` file. This file contains the list of pixel coordinates that make up the path we are interested in:

```
with open("path.p", "rb") as pathFile:  
    path = pickle.load(pathFile)
```

4. Next, we read the header lines from an ASCII Grid file named `path.asc`. These lines contain the geotransform parameters that we'll need for the coordinate transformation. We store these parameters in a list named `gt`:

```
hdr = [getline("path.asc", i) for i in range(1, 7)]  
gt = [float(ln.split(" ")[-1].strip()) for ln in hdr]
```

5. After that, we loop through each set of pixel coordinates in the path, transforming them into real-world coordinates using our `pix2coord` function. We store the resulting sets of real-world coordinates in a list named `coords`:

```
coords = []  
for y,x in path:  
    coords.append(pix2coord(gt, x,y))
```

6. Finally, we create a new shapefile using the `shapefile.Writer` class. We set its shape type to `POLYLINE`, add a field named `NAME`, and record some metadata. We then use the `line` method to write the list of real-world coordinates to the shapefile:

```
with shapefile.Writer("path", shapeType=shapefile.POLYLINE) as  
w:  
    w.field("NAME")  
    w.record("LeastCostPath")  
    w.line([coords])
```

Good work! You have created a program that can automatically navigate through obstacles, based on a set of rules, and exported it to a file that you can display and analyze in a GIS where you can also style the route's appearance however you want!

We only used three rules, but you can add additional restrictions on how the program picks a path by adding other datasets, such as weather or water bodies, or anything else you can imagine.

Now that we understand blazing a path across an arbitrary surface, we'll look at routing through a network.

## Routing along streets

Routing along streets uses a connected network of lines, which is called a graph. The lines in the graph can have impedance values, which discourage a routing algorithm from including them in a route. Examples of impedance values often include traffic volume, speed limit, or even distance. A

key requirement for a routing graph is that all of the lines, known as edges, must be connected. Road datasets that are created for mapping will often have lines whose nodes do not intersect.

In this example, we'll calculate the shortest route through a graph by distance. We'll use a start and end point, which are not nodes in the graph, meaning we'll have to first find the graph nodes that are the closest to our start and destination points.

To calculate the shortest route, we'll use a powerful pure Python graph library called `NetworkX`, which is a general network graphing library that can create, manipulate, and analyze complex networks, including geospatial networks. If `pip` does not install `NetworkX` on your system, then you can find instructions for downloading and installing `NetworkX` for different operating systems at <http://networkx.readthedocs.org/en/stable/>.

You can download the road network and the start and end points, which are located along the U.S. Gulf Coast, as a ZIP file from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_09\\_Asset\\_Files/routing.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_09_Asset_Files/routing.zip). Then, you can follow these steps:

1. First, we import the necessary libraries. We use `NetworkX` for graph analysis, `math` for mathematical operations, `itertools.tee` for iterating through pairs, and `shapefile` for reading and writing shapefiles:

```
import networkx as nx
import math
from itertools import tee
import shapefile
import os
```

2. Next, we define a function called `haversine` that calculates the distance between two points on Earth's surface using the Haversine formula:

```
def haversine(n0, n1):
    x1, y1 = n0
    x2, y2 = n1
    x_dist = math.radians(x1 - x2)
    y_dist = math.radians(y1 - y2)
    y1_rad = math.radians(y1)
    y2_rad = math.radians(y2)
    a = math.sin(y_dist/2)**2 + math.sin(x_dist/2)**2 \
        * math.cos(y1_rad) * math.cos(y2_rad)
    c = 2 * math.asin(math.sqrt(a))
    distance = c * 6371
    return distance
```

3. Then, we define another function, `pairwise`, which takes an iterable object – in this case, a list of points, and returns pairs of adjacent elements:

```
def pairwise(iterable):
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)
```

4. Now, we specify the location of our road network shapefile and create an empty directed graph using NetworkX:

```
shp = "road_network.shp"
G = nx.DiGraph()
```

5. Next, we read the road network shapefile and use its lines to build our graph:

```
r = shapefile.Reader(shp)
for s in r.shapes():
    for p1, p2 in pairwise(s.points):
        G.add_edge(tuple(p1), tuple(p2))
```

6. Then, we extract the largest connected component from our graph. This will be the subgraph on which we will find our route:

```
G_undirected = G.to_undirected()
connected_components = list(nx.connected_components(G_
undirected))
sg = G_undirected.subgraph(connected_components[0])
```

7. After that, we read another shapefile containing our start and end points:

```
r = shapefile.Reader("start_end")
start = r.shape(0).points[0]
end = r.shape(1).points[0]
```

8. Next, we calculate the distance for each edge in our subgraph using the `haversine` function:

```
for n0, n1 in sg.edges():
    dist = haversine(n0, n1)
    sg.edges[n0, n1]["dist"] = dist
```

9. Now, we identify the nodes in our subgraph that are closest to our start and end points:

```
nn_start = None
nn_end = None
start_delta = float("inf")
end_delta = float("inf")
for n in sg.nodes():
```

```

s_dist = haversine(start, n)
e_dist = haversine(end, n)
if s_dist < start_delta:
    nn_start = n
    start_delta = s_dist
if e_dist < end_delta:
    nn_end = n
    end_delta = e_dist

```

- Finally, we find the shortest path between these nodes using Dijkstra's algorithm and save this route as a new shapefile:

```

path = nx.shortest_path(sg, source=nn_start, target=nn_end,
weight="dist")
w = shapefile.Writer(os.path.join(savedir, "route"), shapefile.
POLYLINE)
w.field("NAME", "C", 40)
w.line([[list(p) for p in path]])
w.record("route")
w.close()

```

The following screenshot shows the road network in light gray, the start and end points, and the route in black. You can see that the route cuts across the road network in order to reach the road that is the nearest to the endpoint via the shortest possible distance:

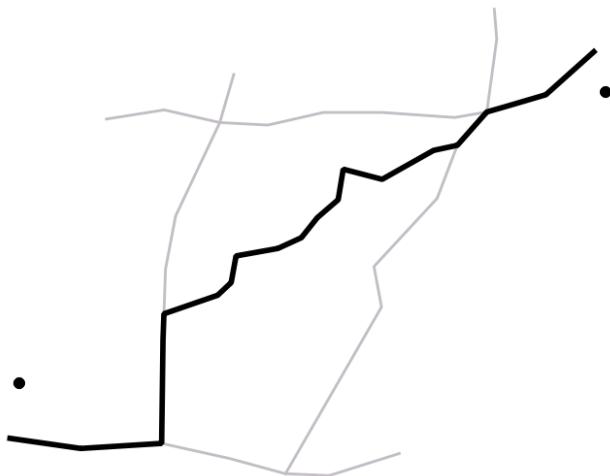


Figure 9.13 – The shortest possible street route between a start and end point

Now that we know how to create various types of routes, we can look at locating photos that you might take while traveling along a route.

## Geolocating photos

Photos that are taken with GPS-enabled cameras, including smartphones, store location information in the header of the file in a format called **EXIF** tags. These tags are based largely on the same header tags that are used by the TIFF image standard. In this example, we'll use those tags to create a shapefile with point locations for photos and file paths to the photos, as attributes.

We'll use the **PIL** in this example because it has the ability to extract EXIF data. Most photos that are taken with smartphones are geotagged images; however, you can download the set used in this example from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_09\\_Asset\\_Files/photos.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_09_Asset_Files/photos.zip):

1. First, we import the necessary libraries. We use `glob` for file pattern matching, `os` for operating system interaction, `Image` for image manipulation, and `shapefile` for dealing with shapefiles:

```
import glob
import os
try:
    import Image
    import ImageDraw
except:
    from PIL import Image
    from PIL.ExifTags import TAGS
import shapefile
```

2. Next, we define a function, `exif`, which returns EXIF metadata from a given image. EXIF metadata often includes GPS information:

```
def exif(img):
    exif_data = {}
    try:
        i = Image.open(img)
        tags = i._getexif()
        for tag, value in tags.items():
            decoded = TAGS.get(tag, tag)
            exif_data[decoded] = value
    except:
        pass
    return exif_data
```

3. Then, we define a function, `dms2dd`, to convert coordinates from **degrees, minutes, and seconds (DMS)** to **decimal degrees (DD)**:

```
def dms2dd(d, m, s, i):
    sec = float((m * 60) + s)
```

```
dec = float(sec / 3600)
deg = float(d + dec)
if i.upper() == 'W':
    deg = deg * -1
elif i.upper() == 'S':
    deg = deg * -1
return float(deg)
```

- After that, we define a function, `gps`, to extract GPS information from the EXIF metadata:

```
def gps(exif):
    lat = None
    lon = None
    if exif['GPSInfo']:
        coords = exif['GPSInfo']
        i = coords[1]
        d = coords[2][0]
        m = coords[2][1]
        s = coords[2][2]
        lat = dms2dd(d, m, s, i)
        i = coords[3]
        d = coords[4][0]
        m = coords[4][1]
        s = coords[4][2]
        lon = dms2dd(d, m, s, i)
    return lat, lon
```

- Now, we initialize a dictionary to hold the filenames and GPS coordinates of photos. If possible, give the `photos` directory the shortest possible absolute path to make it easy to test this shapefile out in QGIS:

```
photos = {}
photo_dir = "/photos"
```

- Next, we use `glob` to find all JPEG photos in the specified directory:

```
files = glob.glob(os.path.join(photo_dir, "*.jpg"))
```

- Then, we loop through each file, extract its EXIF metadata, and store its GPS coordinates in the `photos` dictionary:

```
for f in files:
    e = exif(f)
    lat, lon = gps(e)
    photos[f] = [lon, lat]
```

8. Finally, we create a point shapefile to store the locations of these photos, adding the filename as an attribute:

```
with shapefile.Writer("photos", shapefile.POINT) as w:
    w.field("NAME", "C", 80)
    for f, coords in photos.items():
        w.point(*coords)
        w.record(f)
```

The filenames of the photos in the shapefile are now attributes of the point locations where the photos were taken. GIS programs including QGIS and ArcGIS have the tools to turn those attributes into links when you click on the photo path or the point. The following screenshot from QGIS shows that one of the photos opens after clicking on the associated point using the Run Feature Action tool:

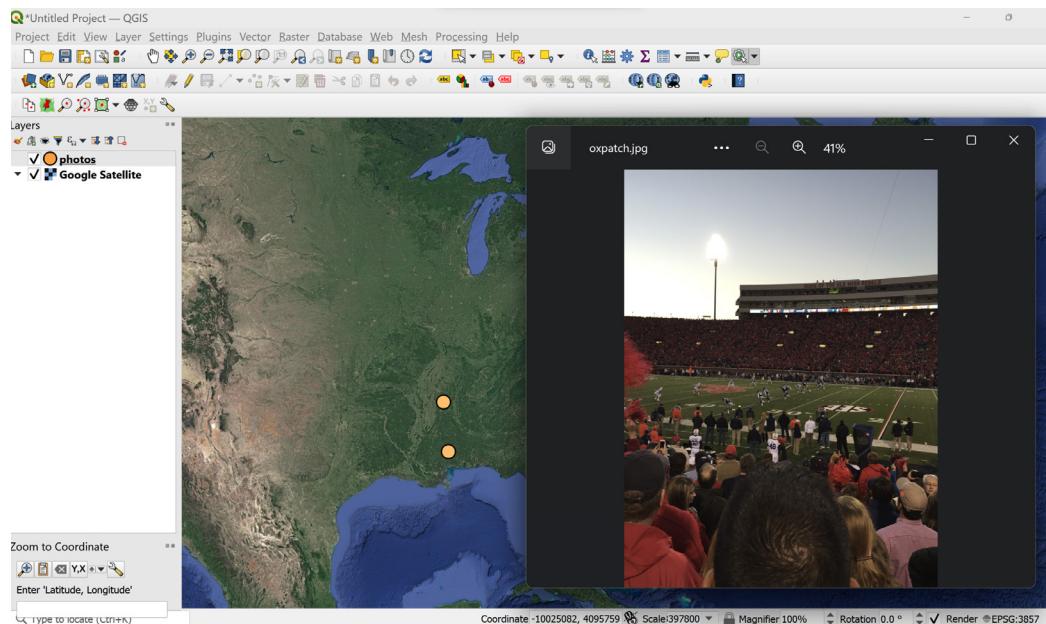


Figure 9.14 – A geolocated image popup in QGIS

To view the result, please use the following instructions:

1. Download QGIS 3.30 or higher from <https://qgis.org> and follow the installation instructions.
2. Open QGIS and drag the `photos.shp` file onto the blank map.
3. In the **Layer** panel on the left, right-click on the layer named **Photos**, hover over **Set Layer CRS**, and then choose **EPSG:4326 – WGS 84** to set the projection.
4. Right-click on the layer again and this time choose **Properties**.

5. On the **Actions** tab, click the green plus sign to open the new actions dialog.
6. In the **Type** drop-down menu, select **Open URL**.
7. In the **Description** field, enter **Open Image**.
8. Click on the **Insert** button in the lower-right corner.
9. Click the **OK** button, and then close the properties dialog.
10. Click on the small black arrow to the right of the **Run Feature Action** tool, which is a gear icon with a green center and a white arrow in it.
11. In the menu that pops up, choose **Open Image**.
12. Now, click on one of the points on the map to see the geotagged image pop up. If it doesn't, ensure the path to the photos you set in the script is an absolute path from the root directory of your computer. If it's not, remove the layer from QGIS, set the path to an absolute path, and repeat the process.

Now, let's move from an image taken on Earth to images taken of Earth itself, by working with satellite images.

## Calculating satellite image cloud cover

Satellite images give us a powerful bird's-eye view of Earth. They are useful for a variety of purposes, which we saw in *Chapter 7, Python and Remote Sensing*. However, they have one flaw—clouds. As a satellite passes around Earth and collects imagery, it inevitably images clouds. And in addition to obstructing our view of Earth, the cloud data can adversely affect remote sensing algorithms by wasting CPU cycles on useless cloud data, or skew the results by introducing unwanted data values.

The solution is to create a cloud mask. A cloud mask is a raster that isolates the cloud data in a separate raster. You can then use that raster as a reference when processing the image in order to avoid cloud data, or you can even use it to remove the clouds from the original image.

In this section, we'll create a cloud mask for a Landsat image using the `rasterio` module and the `rio-18qa` plugin. The cloud mask will be created as a separate image that just contains clouds:

1. First, we need to download some sample Landsat 8 satellite image data as a ZIP file from [https://drive.google.com/file/d/1hzI470qBplIM34Eu76ccnDeH8eemptyT/view?usp=drive\\_link](https://drive.google.com/file/d/1hzI470qBplIM34Eu76ccnDeH8eemptyT/view?usp=drive_link)
2. Click the download icon at the top right to download the data as a ZIP file, and unzip it to a directory named `L8`.
3. Next, make sure you have the raster libraries that we need by running `pip`:

```
pip install rasterio -U
pip install rio-18qa
```

4. We import the necessary libraries. We use `glob` to search for files, `os` to interact with the operating system, `rasterio` for raster data manipulation, and a specific function, `write_cloud_mask` from the `18qa.qa` package, to create the cloud mask:

```
import glob
import os
import rasterio
from 18qa.qa import write_cloud_mask
```

5. Next, we specify the directory where our Landsat 8 data resides. We assume all relevant files are in a folder named `L8`:

```
landsat_dir = "L8"
```

6. Then, we look for the file containing **quality assurance (QA)** metadata within our `L8` directory. We use the `glob` library to search for files matching the pattern `*QA*`, and we grab the first one that comes up:

```
src_qa = glob.glob(os.path.join(landsat_dir, '*QA*'))[0]
```

7. Finally, we read the QA data into a raster object using `rasterio`. We also update the profile metadata to specify that the `nodata` value should be 0. With this setup, we call the `write_cloud_mask` function to create and save the cloud mask as `cloudmask.tif`:

```
with rasterio.open(src_qa) as qa_raster:
    profile = qa_raster.profile
    profile.update(nodata=0)
    write_cloud_mask(qa_raster.read(1), profile, 'cloudmask.tif')
```

The following figure shows just the band 7 (short-wave infrared) image from the Landsat 8 dataset:

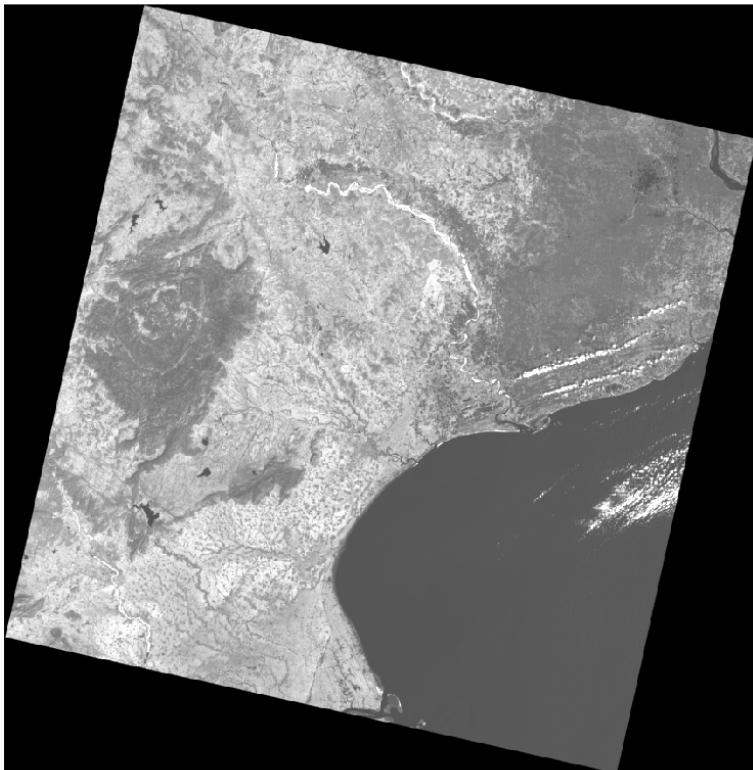


Figure 9.15 – The Landsat 8 image with clouds

The next figure shows the cloud mask image containing only the location of clouds and shadows:

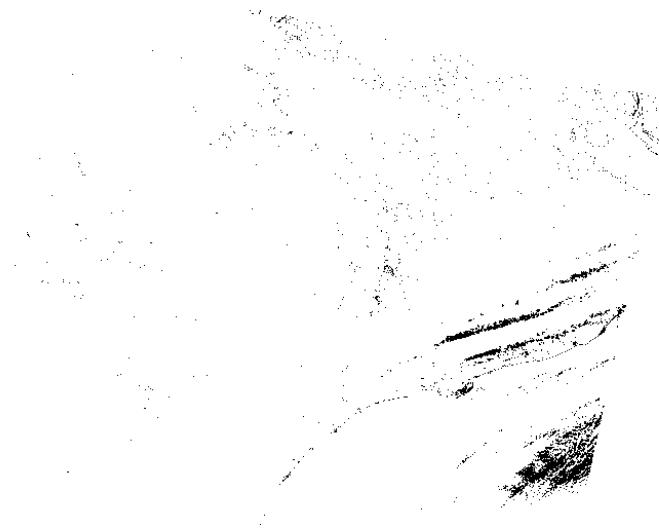


Figure 9.16 – The cloud mask image we generated

And finally, here's the mask over the image, showing the clouds as black:

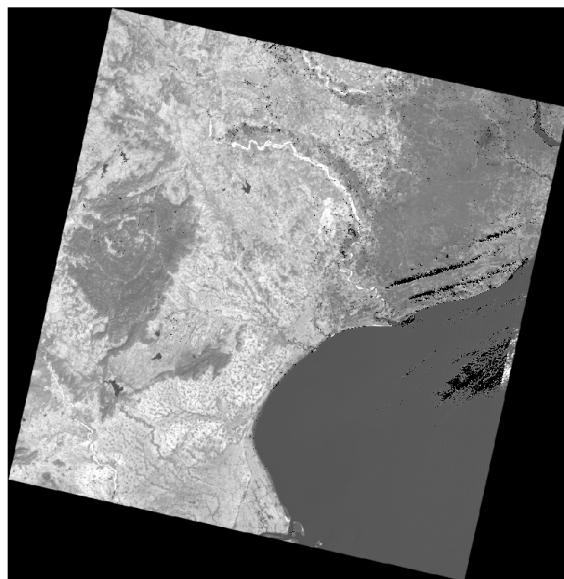


Figure 9.17 – The Landsat satellite image with the cloud mask overlayed

This example brushes the surface of what you can do with image masking. Another `rasterio` module, `rio-cloudmask`, allows you to calculate the cloud mask from scratch without using the quality-assurance data. But it requires some additional pre-processing steps. You can learn more about that here: <https://github.com/mapbox/rio-cloudmask>.

## Summary

In this chapter, we learned how to create an NDVI crop health index, a flood inundation model, and an optimal terrain route, which are used every day in government, science, and industry. We were able to use minimal, free cross-platform Python tools instead of expensive commercial GIS and remote sensing tools. In addition to the examples in this chapter, you now have some more reusable functions, algorithms, and processing frameworks for other advanced analyses, which will allow you to solve new problems that you come across in fields such as transportation, agriculture, and weather.

In the next chapter, we'll move into a relatively new area of geospatial analysis: real-time and near-real-time data.

## Questions

1. What is the formula for a normalized differential vegetation index (NDVI) that shows relative plant health?
2. Which model in this chapter would you most likely use if you were a hydrologist?
3. What are the two major algorithms used for routing?

## Further reading

- The **Sea Level Rise Viewer** is an interactive sea-level rise tool from NOAA lets you use a slider on the left side of the map to raise the water level in the United States rise to inundate coastal areas. You can zoom and pan the map for views at different locations and scales: <https://coast.noaa.gov/slrv/>



# 10

## Working with Real-Time Data

A common saying among geospatial analysts is, “*A map is outdated as soon as it’s created.*” This saying reflects the fact that the Earth and everything on it are constantly changing. For most of the history of geospatial analysis and through most of this book, geospatial products are relatively static. Raw datasets are typically updated anywhere from a few months to a few years. The age of geospatial data in a map is referred to as **data currency**.

Data currency has traditionally not been the primary focus because of the time and expense needed to collect data. Web mapping, wireless cellular modems, and low-cost GPS antennas have changed that focus. It is now logically feasible and even quite affordable to monitor a rapidly changing object or system and broadcast those changes to millions of people online. This change is revolutionizing geospatial technology and taking it in new directions. The most direct evidence of this revolution is web mapping mashups that use systems such as Google Maps or OpenLayers and web-accessible data formats. Every day, more and more electronic devices are being brought online to broadcast their location and data for automation or remote control. Examples include thermostats, cameras, and cars. You can also use cheap, embedded computers such as the popular Raspberry Pi to turn almost anything into a connected smart device. This concept of connecting devices into a web of data and information is called the **Internet of Things (IoT)**.

In this chapter, we’ll be checking out the following topics:

- Limitations of real-time data
- Using real-time data
- Tracking vehicles
- Gathering reports from the field

By the end of this chapter, you’ll have learned how to work with real-time geospatial data and will be able to build a field reporting tool that can serve as a data transmission source for any type of data.

## Technical requirements

We will need the Google GTFS Python library for this chapter. You can install it with pip:

```
pip install --upgrade gtfs-realtime-bindings
```

The code files for this chapter are available at [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730\\_10\\_Asset\\_Files](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730_10_Asset_Files).

## Limitations of real-time data

The term **real-time data** typically means near-real-time. Some tracking devices capture real-time data and may update as often as several times a second. However, the limitations of the infrastructure that broadcasts that data may constrain the output to every 10 seconds or longer. Weather radar is a perfect example. A **Doppler Weather Radar (DWR)** sweeps continuously but data is typically available online every 5 minutes. But given the contrast with traditional geospatial data updates, a refresh of a few minutes is real-time enough. Its limitations can be summarized as follows:

- Network bandwidth limitations restricting data size
- Network latency limiting the data update frequency
- Availability of the data source due to restrictions such as battery life
- Lack of quality control due to data being instantly available to consumers
- Security vulnerabilities due to the rapid ingestion of unverified data

Real-time data opens up additional opportunities for geospatial applications, so we'll look at using it next.

## Using real-time data

Web mashups often use real-time data. Web mashups are essentially digital compositions that combine data, services, or functionalities from multiple sources into a single integrated experience. Think of them as the DJ mixes of the web world, where you're blending Google Maps with weather data, for instance, to create a new, more informative map that shows weather patterns over geographic locations. Web mashups are amazing and have changed the way many different industries operate. However, they are typically limited in that they usually just display pre-processed data on a map and give developers access to a JavaScript API. But what if you want to process the data in some way? What if you want to filter, change, and then send it to another system? To use real-time data for geospatial analysis, you need to be able to access it as point data or a georeferenced raster.

### More about web mashups

You can find out more about web map mashups (also known as digital map mashups) here: <https://www.esri.com/arcgis-blog/products/product/uncategorized/digital-map-mashups/>

As with the examples in the previous chapters, the scripts are as simple as possible and designed to be read from start to finish without much mental looping. When functions are used, they are listed first, followed by script variable declarations, and finally the main program execution.

Now, let's learn how to access a real-time and point-location data source using vehicles through the **General Transit Feed Specification (GTFS)** feed.

## Tracking vehicles

For our first real-time data source, we'll use the excellent **Washington Metropolitan Area Transit Authority (WMATA)** real-time GTFS feed. GTFS, originally developed by Google, is a standardized data format that public transportation agencies use to share scheduling and geographical information. Imagine it as the DNA code for a transit system, detailing when and where buses or trains stop, the routes they follow, and even fare structures. This data is often used in various applications and services to provide real-time transit information to the public.

### Getting API keys

The examples in this chapter require free API keys for the different services used. Sample keys are provided for testing, but they are limited, so you are strongly encouraged to register with each of these data services and obtain your own key.

## Getting a vehicle location

We'll use Google's GTFS library to access the WMATA real-time bus feed and track the location of a bus. We'll ask WMATA for the location of all of its buses and just grab the coordinates of the first one. The steps are as follows:

1. First, import the necessary modules. These modules help us interact with GTFS data and make web requests:

```
from google.transit import gtfs_realtime_pb2  
from urllib.request import Request, urlopen
```

2. Next, initialize a FeedMessage object from the `gtfs_realtime_pb2` module. This object will hold the real-time transit data:

```
feed = gtfs_realtime_pb2.FeedMessage()
```

3. Then, set up a web request to the GTFS Realtime API. We must specify the URL and add an API key in the header:

```
req = Request('https://api.wmata.com/gtfs/bus-gtfsrt-  
vehiclepositions.pb')  
req.add_header('api_key', \  
'd4d713a59a9f477fab8c1e9fbadbd467')
```

4. After that, execute the web request and store the response:

```
response = urlopen(req)
```

5. Now, parse the response to populate our `FeedMessage` object with real-time data:

```
feed.ParseFromString(response.read())
```

6. Subsequently, extract the first bus entity from the feed. This gives us information about a specific bus:

```
bus = feed.entity[0]
```

7. Then, grab the bus ID, latitude, and longitude from the bus entity:

```
bus_id = bus.id  
bus_lat = bus.vehicle.position.latitude  
bus_lon = bus.vehicle.position.longitude
```

8. Finally, print out the bus's ID along, with its current latitude and longitude:

```
print(f'Bus {bus_id} is at latitude: {bus_lat}, longitude: {bus_  
lon}')
```

You should see something similar to the following:

```
Bus 8091 is at latitude: 38.84510040283203, longitude:  
-76.98832702636719
```

Now that we can locate a bus, let's put the location on a map.

## Mapping a vehicle location

We'll need a dynamic map service that lets us plot street locations anywhere in the world. For that, we'll use the MapQuest API, which provides a free mapping service for applications like this. Mapquest will put a simple marker on the map for the bus's location. The map will update every few seconds or however long we set in the script. We'll turn our previous script into a function and then create a new function to get the map:

1. First, let's import the required modules. These include modules for GTFS real-time data, web requests, XML parsing, and time management:

```
from google.transit import gtfs_realtime_pb2
from urllib.request import Request, urlopen
from xml.dom import minidom
import time
```

2. Next, define a function called `get_bus`. This function fetches the most recent latitude and longitude of a bus using the GTFS real-time API:

```
def get_bus():
    bus_lat = False
    bus_lon = False
    feed = gtfs_realtime_pb2.FeedMessage()
    req = Request('https://api.wmata.com/gtfs/bus-gtfsrt-
vehiclepositions.pb')
    req.add_header('api_key', \
        'd4d713a59a9f477fab8c1e9fbadbd467')
    response = urlopen(req)
    feed.ParseFromString(response.read())
    bus = feed.entity[0]
    bus_lat = bus.vehicle.position.latitude
    bus_lon = bus.vehicle.position.longitude
    return (bus_lat, bus_lon)
```

3. Then, define another function called `bus_map`. This function uses the OpenStreetMap static map API to plot the bus location on a map and save it as an image:

```
def bus_map(mapimg):
    lat, lon = get_bus()
    print(f"Located bus at {lon}, {lat}")
    if not lat:
        return False
    mqapi = "https://www.mapquestapi.com/staticmap/v5/map?"
    mqapi += "locations={},{}&".format(lat, lon)
    mqapi += "zoom=18&"
```

```
mqapi += "&size=1500,1000"
mqapi += "&key=7iWl6hsFNcm1GSojIU77zZR39W4GJKHP"
img = urlopen(mqapi)
with open("{}{}.png".format(mapimg), "wb") as f:
    f.write(img.read())
return True
```

- After that, set some variables for the main loop. These include the name of the map image to save, the number of updates we want, and the frequency of updates:

```
nextimg = "bus_map"
requests = 3
freq = 5
```

- Finally, enter the main loop. Here, we call the bus\_map function to update the bus location on the map and save it as an image. We do this every few seconds, as specified by the freq variable:

```
for i in range(requests):
    success = bus_map(nextimg)
    if not success:
        print("No data available.")
        continue
    print(f"Saved map {i} at {time.asctime()}")
    time.sleep(freq)
```

- While the script runs, you'll see output similar to the following:

```
Located bus at -77.0427017211914, 38.92279052734375
Saved map 0 at Tue Sep 12 19:35:08 2023
Located bus at -77.0427017211914, 38.92279052734375
Saved map 1 at Tue Sep 12 19:35:14 2023
Located bus at -77.0427017211914, 38.92279052734375
Saved map 2 at Tue Sep 12 19:35:20 2023
```

This script will output images similar to the following:

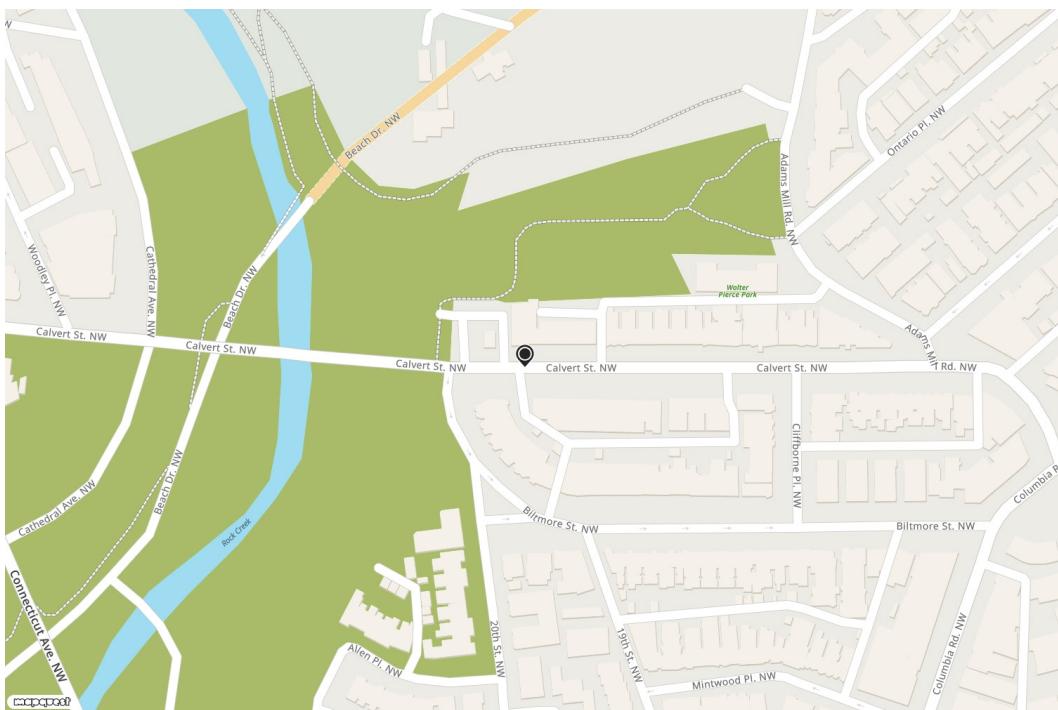


Figure 10.1 – The bus’s location plotted on a map with the MapQuest StaticMap API

This map is an excellent example of using an API to create a custom mapping product, but it is a very basic tracking application. To begin to develop it into a more interesting geospatial product, we need to combine it with some other real-time data source that gives us more situational awareness.

Now that we can track buses, let’s add some additional information to the map that would be useful to know for passengers taking a bus somewhere. Let’s add some weather data.

## Storm chasing

We can enhance our bus map by adding weather data to see if there’s any precipitation on the bus route. The **National Oceanic and Atmospheric Administration (NOAA)** provides a variety of web-based weather services that you can incorporate into geospatial applications. For our bus map, we’ll use their NEXRAD precipitation WMS service. Instead of the Mapquest API, this time, we’ll use the OpenStreetMap WMS service, which allows us to use a single function for both the weather and street map services. Because weather data can be unpredictable, our weather image may be blank. We’ll have the script detect blank images and print a message on the map stating that there’s no precipitation in the area. Instead of a simple map marker, this time, we’ll incorporate a bus icon, which you can download from <https://github.com/PacktPublishing/Learning-Geospatial->

Analysis-with-Python-Fourth-Edition/blob/main/B19730\_10\_Asset\_Files/busicon.png?raw=true:



Figure 10.2 – Bus icon map marker

Because we are going to blend the street map and weather map, we'll need the **Python Imaging Library (PIL)** that we used in previous chapters. We'll replace our `bus_map()` function from the previous example with a simple `wms()` function that can grab a map image via a bounding box from any WMS service. We'll also add a function that converts decimal degrees into meters, named `ll2m()`.

The script gets the bus location, converts the location into meters, creates a 2-mile (3.2 km) rectangle around the location, and then downloads a street and weather map. The map images are then blended using PIL. After this, PIL shrinks the bus icon image to 30 x 30 pixels and pastes it in the center of the map, which is the bus's location. Let's look at how the code works:

1. First, import the necessary modules for handling GTFS real-time data, web requests, XML parsing, and image manipulation:

```
import sys
from google.transit import gtfs_realtime_pb2
from urllib.request import Request, urlopen
from xml.dom import minidom
import math
try:
    import Image
except:
    from PIL import Image
```

2. Next, define a function called `get_bus` to fetch the most recent latitude and longitude details of a bus using the GTFS real-time API:

```
def get_bus():
    bus_lat = False
    bus_lon = False
    feed = gtfs_realtime_pb2.FeedMessage()
    req = Request('https://api.wmata.com/gtfs/bus-gtfsrt-
vehiclepositions.pb')
    req.add_header('api_key', \
'd4d713a59a9f477fab8c1e9fbadbd467')
    response = urlopen(req)
    feed.ParseFromString(response.read())
    bus = feed.entity[0]
    bus_lat = bus.vehicle.position.latitude
    bus_lon = bus.vehicle.position.longitude
    return (bus_lat, bus_lon)
```

3. Then, define a function called `ll2m` to convert latitude and longitude into meters, which is useful for mapping:

```
def ll2m(lon, lat):
    x = lon * 20037508.34 / 180.0
    y = math.log(math.tan((90.0 + lat) * math.pi / 360.0))\
/ (math.pi / 180.0)
    y = y * 20037508.34 / 180
    return (x, y)
```

4. After that, define a function called `wms` to retrieve a WMS map image from a specified service and save it as a PNG:

```
def wms(minx, miny, maxx, maxy, service, lyr, img, w, h):
    wms = service
    wms += "?SERVICE=WMS&VERSION=1.1.1&REQUEST=GetMap&"
    wms += "LAYERS=%s" % lyr
    wms += "&STYLES=&"
    wms += "SRS=EPSG:900913&"
    wms += "BBOX=%s,%s,%s,%s" % (minx, miny, maxx, maxy)
    wms += "WIDTH=%s&" % w
    wms += "HEIGHT=%s&" % h
    wms += "FORMAT=image/png"
    wmsmap = urlopen(wms)
    with open(img + ".png", "wb") as f:
        f.write(wmsmap.read())
```

5. Subsequently, set up some variables, such as the WMS services we'll be using, the image dimensions, and the icon size:

```
basemap = "https://ows.terrestris.de/osm/service"
streets = "OSM-WMS"
mapimg = "basemap"
weather = "https://opengeo.ncep.noaa.gov/geoserver/conus/conus_
pcpn_typ/ows"
weather_layer = "conus_pcpn_typ"
skyimg = "weather"
final = "next-weather"
opacity = .5
w = 1000
h = 1000
icon = 30
```

6. Next, fetch the bus's location using the `get_bus` function:

```
lat, lon = get_bus()
if not lat:
    print("No bus data available.")
    print("Please try again later")
    sys.exit()
```

7. Then, convert the latitude and longitude into Web Mercator coordinates:

```
lat = float(lat)
lon = float(lon)
x, y = ll2m(lon, lat)
```

8. Now, create a bounding box around the bus's location and download the street map and weather map using the `wms` function:

```
minx = x - 1600
maxx = x + 1600
miny = y - 1600
maxy = y + 1600
wms(minx, miny, maxx, maxy, basemap, streets, mapimg, w, h)
wms(minx, miny, maxx, maxy, weather, weather_layer, \
skyimg, w, h)
```

9. Finally, blend the base map and weather map, place a bus icon at the bus's location, and save the final map:

```
im1 = Image.open("basemap.png").convert('RGBA')
im2 = Image.open("weather.png").convert('RGBA')
im3 = Image.blend(im1, im2, opacity)
```

```
im4 = Image.open("busicon.png")
im4.thumbnail((icon, icon))
w, h = im3.size
w2, h2 = im4.size
center_width = int((w/2)-(w2/2))
center_height = int((h/2)-(h2/2))
im3.paste(im4, (center_width, center_height), im4)
im3.save(final + ".png")
```

The script will produce an image similar to the following:

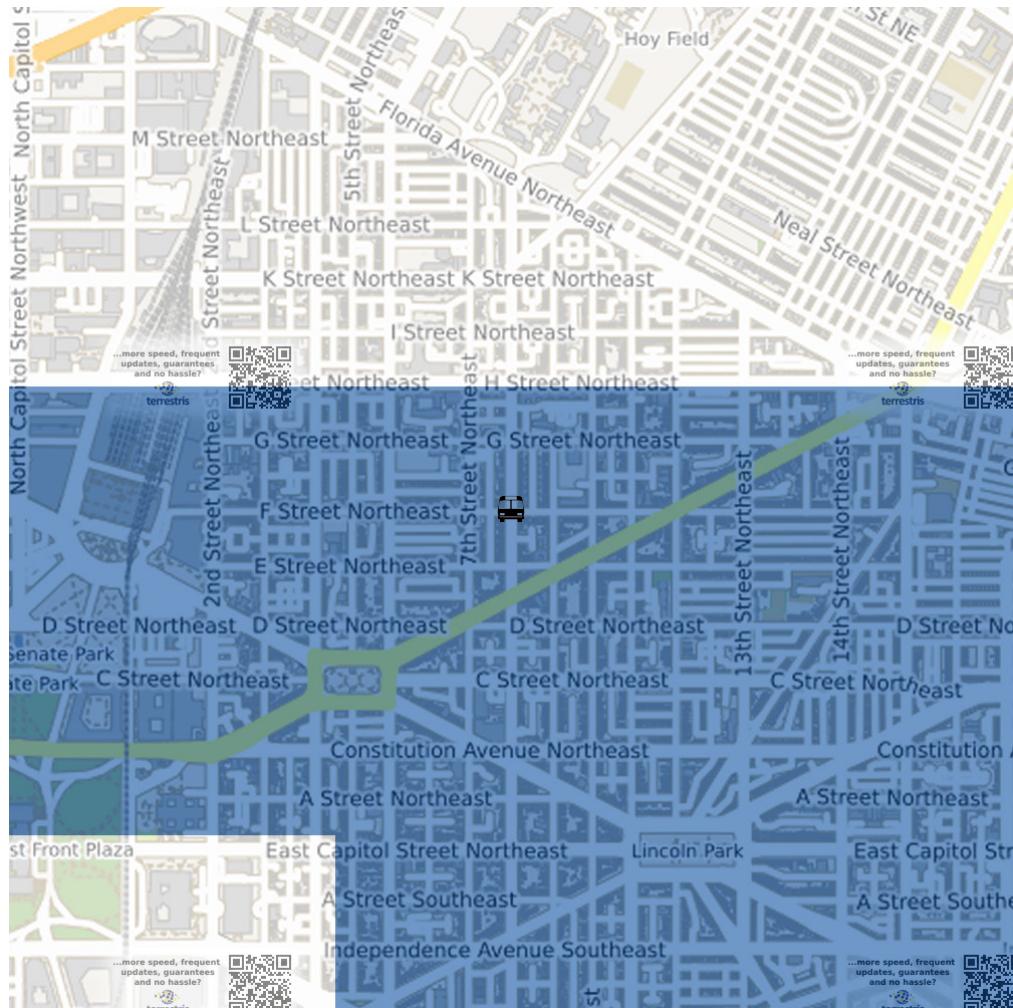


Figure 10.3 – Real-time bus map with a precipitation layer

This map shows us that the bus is experiencing moderate precipitation at its current location. The color ramp ranges from light blue for light precipitation, then green, yellow, orange, to red as the rain gets heavier (or light gray to darker gray in black and white). So, at the time this map was created, the bus-line operator could use this image to tell their drivers to go a little slower, and passengers would know they may want to get an umbrella before heading to the bus stop.

Now that we've learned how to check the weather, let's combine discrete real-time data sources into more meaningful products using Python, HTML, and JavaScript.

## Gathering reports from the field

In our final example in this chapter, we'll get off the bus and out into the field. Smartphones, tablets, and laptops allow us to update a GIS and view those updates from anywhere. We'll use HTML, GeoJSON, the Leaflet JavaScript library, and a pure-Python library named Folium to create a client-server application that allows us to post geospatial information to a server and then create an interactive web map to view those data updates.

First, we need a web form that shows your current location and updates the server when you submit the form with comments about your location. You can find the form here: <http://geospatialpython.github.io/Learn/LGAWPFieldWork.html>.

The following screenshot shows the form:

The screenshot shows a web-based field data collection form. At the top, it says "Field Data Collection Form". Below that is a map of a city area with a blue circle centered on a location marked with a pin. The map includes street names like Jourdan River, Blue Meadow Road, Dunbar Avenue, and Felicity Street. In the bottom left corner of the map area, there is a small note: "Leaflet | Map data © OpenStreetMap contributors, CC-BY-SA, Imagery © Mapbox". Below the map is a text box containing instructions: "This is the real-time field data collection example for the book 'Learning Geospatial Analysis with Python'. Allow the browser to get your location, fill out the form below, and click submit. Then follow the instructions in the book to turn the data into a Leaflet map." At the very bottom of the form are four input fields: "Name:" with a text input, "Comment:" with a text input, "Date:" with a date input (mm/dd/yyyy) and a checkbox, and a "Submit" button.

Figure 10.4 – Field data collection form

You can view the source of that form to see how it works. The mapping is done using the Leaflet library and posts GeoJSON to a unique URL on [jsonbin.io](http://jsonbin.io). You can use this page on mobile devices, move it to any web server, or even use it on your local hard drive.

Next, you need to install the `folium` library from PyPI if you haven't already. Folium provides a simple Python API for creating Leaflet web maps. You can find more information about Folium here: <https://github.com/python-visualization/folium>.

Folium makes producing a Leaflet map extremely simple. This script is just a few lines long and will output a web page named `map.html`. We will pass the GeoJSON URL to the `map` object, which will plot the locations on the map. In the following steps, we'll plot the field reports on a Folium map:

1. First, import the required libraries – `folium` for map rendering, `requests` for HTTP requests, and `JSON` for handling JSON data:

```
import folium
import requests
import json
```

2. Next, define a function called `read_bin` to fetch data from a `JSONBin.io` bin using an API key and bin ID:

```
def read_bin(api_key, bin_id):
    url = f"https://api.jsonbin.io/v3/b/{bin_id}/latest"
    headers = {
        "X-Master-Key": api_key
    }
    response = requests.get(url, headers=headers)

    if response.status_code == 200:
        return response.json()["record"]
    else:
        print(f"Failed to read bin: {response.status_code}")
        print(response.json())
        return None
```

3. Then, define a function called `add_geojson_to_map` to add GeoJSON data to a Folium map object. This function also sets the style and popup fields for the GeoJSON layer:

```
def add_geojson_to_map(geojson, map_obj):
    def style_function(feature):
        return {'fillColor': '#ffaf00', 'color': 'blue',
'weight': 1.5}
    folium.GeoJson(
        geojson,
        style_function=style_function,
        popup=folium.GeoJsonPopup(fields=list(geojson['features'][0]
['properties'].keys())))
    ).add_to(map_obj)
```

- After that, define a function called `get_bounds` to extract the bounding coordinates from the GeoJSON data. This will help us fit the map view to these bounds:

```
def get_bounds(geojson):  
    coordinates = []  
    for feature in geojson['features']:  
        coords = feature['geometry']['coordinates'][0]  
        coordinates.append([coords[1], coords[0]]) # Flip to  
        [lat, lon]  
    return coordinates
```

- Subsequently, set up our API key and bin ID for `JSONBin.io`, and initialize a Folium map object:

```
api_key = "$2b$10$lypZoIQPtYtz1PTSk75KjuzUGMnupW1pSJdqtU.  
wSmxuGZTDjIpy" # Replace with your actual JSONBin.io API key  
bin_id = "64fe6ae18d92e126ae6a1a23"  
m = folium.Map()
```

- Next, read the GeoJSON data from the `JSONBin.io` bin using the `read_bin` function:

```
geojson = read_bin(api_key, bin_id)
```

- Then, add this GeoJSON data to our Folium map using the `add_geojson_to_map` function:

```
add_geojson_to_map(geojson, m)
```

- Finally, fit the map view to the bounds of the GeoJSON data using the `get_bounds` function and save the map as an HTML file:

```
bounds = get_bounds(geojson)  
m.fit_bounds(bounds)  
m.save("map.html")
```

If you open the `map.html` file in a browser, you'll see the GeoJSON feed on a map. Here, you can click on the point markers to see the properties from the input form, similar to what's shown here:

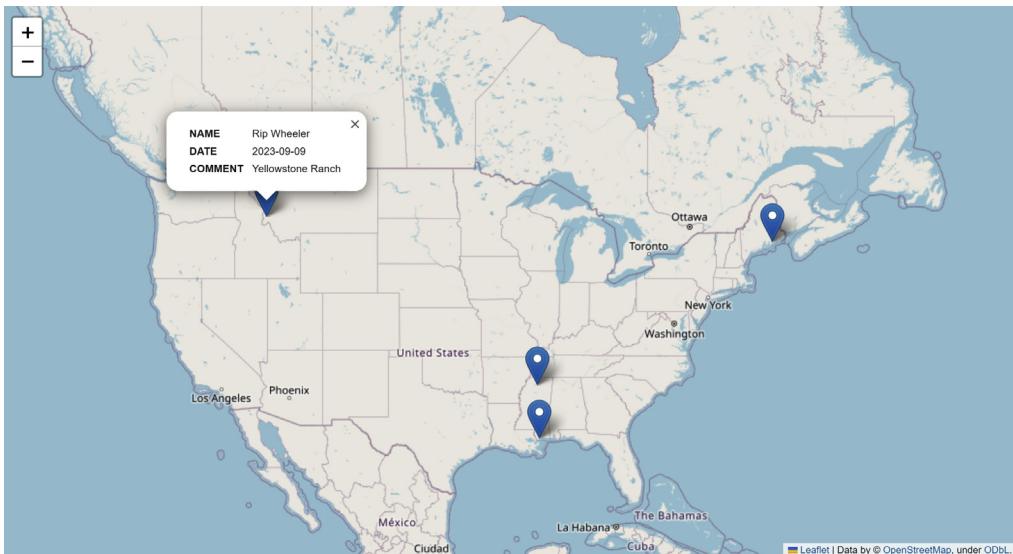


Figure 10.5 – The field data collection map

The form has been set up to limit the number of entries in the JSON store to 5 points to prevent the file from growing too big.

## Summary

Real-time data is an exciting way to do new types of geospatial analysis, and it was only recently made possible by advances in several different technologies, including web mapping, GPS, and wireless communications. In this chapter, you learned how to access raw feeds for real-time location data, how to acquire a subset of real-time raster data, how to combine different types of real-time data into a custom map analysis product using only Python, and how to build client-server geospatial applications to update a GIS in real time.

As with the previous chapters, this chapter's examples contained building blocks that will let you build new types of applications using Python that go far beyond the typical popular and ubiquitous JavaScript-based mashups.

In the next chapter, we will combine everything we've learned so far into a complete geospatial application that applies algorithms and concepts in a realistic scenario.

## Questions

1. What is the term that refers to the age of geospatial data?
2. What is the standard called for transmitting transportation data?
3. What is the term for combining multiple web data sources into a single application?

## Further reading

- This chapter only scraped the surface of the very rich GTFS. Google has a complete reference and guide with additional code samples, which you can find at <https://developers.google.com/transit/gtfs-realtime>.

# 11

## Putting It All Together

Throughout the book, we've touched upon all the important aspects of geospatial analysis, and we've used a variety of different techniques in Python to analyze different types of geospatial data. In this final chapter, we will draw on nearly all of the topics we have covered to produce a real-world product that has become very popular – a GPS route analysis report.

These reports are common to dozens of mobile app services, GPS watches, in-car navigation systems, and other GPS-based tools. A GPS typically records location, time, and elevation. From these values, we can derive a vast amount of ancillary information about what happened along the route on which that data was recorded. Fitness apps, including Runkeeper, MapMyRun, Strava, and Nike Plus, all use similar reports to present GPS-tracked exercise data from running, hiking, biking, and walking.

We will create one of these reports using Python. This program is nearly 500 lines of code, our longest yet, so we will step through it in sections. We will combine the following techniques:

- Understanding a typical GPS report
- Building a GPS reporting tool

As we step through this program, all of the techniques used will be familiar, but we will use them in new ways.

### Technical requirements

We will need to install several libraries, which you can install with `pip`:

- The SRTM elevation library:

```
pip install SRTM.py
```

- The pygooglecharts library:

```
pip install pygooglecharts
```

- The meteostat weather library:

```
pip install meteostat
```

- The fpdf library to create PDFs:

```
pip install fpdf
```

The code files for this chapter are on GitHub: [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730\\_11\\_Asset\\_Files](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/tree/main/B19730_11_Asset_Files).

## Understanding a typical GPS report

A typical GPS report has common elements, including a route map, elevation profile, and speed profile. The following screenshot is a report from a typical route logged through Runkeeper (<https://runkeeper.com>):

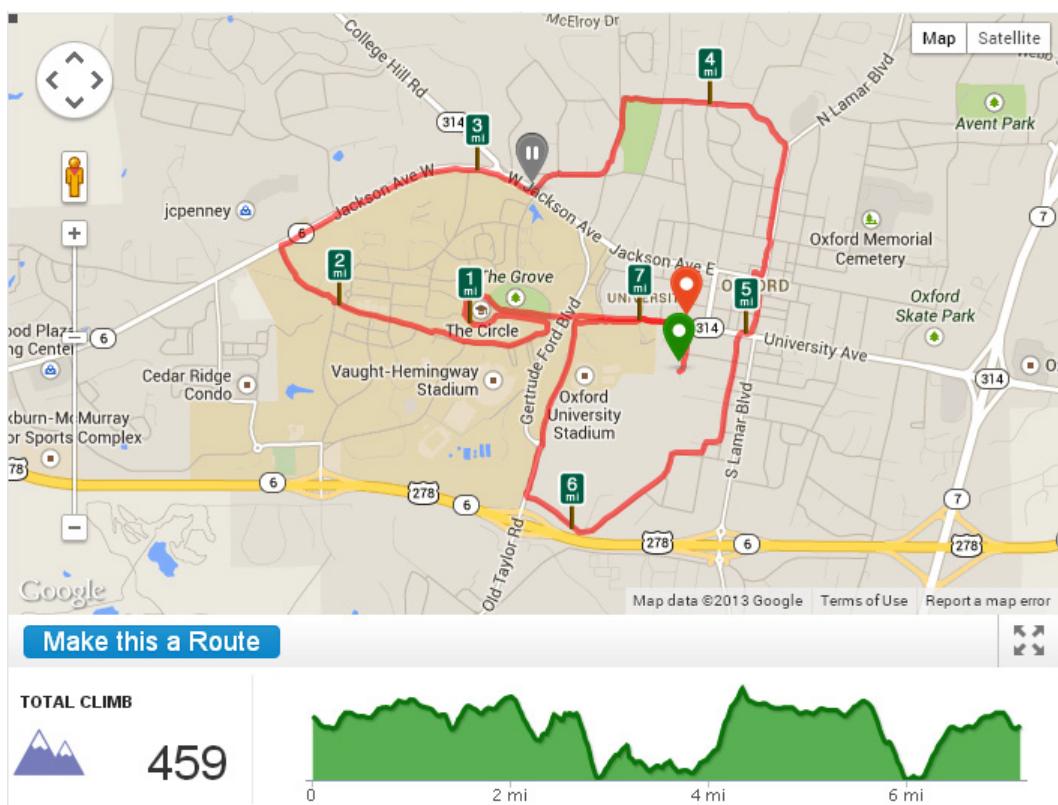


Figure 11.1 – A typical GPS route report from a fitness app

Our report will be similar, but we'll add a twist. We'll include a route map and elevation profile, like this service, but we'll also add the weather conditions that occurred on that route when it was recorded and a geolocated photo taken on the route.

Now that we know what a GPS report is, let's learn how to build it.

## Building a GPS reporting tool

The name of our program is `GPX-Reporter.py`. If you recall from the *Tag-based and markup-based formats* section in *Chapter 2, Learning about Geospatial Data*, the GPX format is the most common way to store GPS route information. Nearly every program and device relying on GPS data can convert to and from GPX.

For this example, you can download a sample GPX file and a photo directory as a ZIP file from [https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730\\_11\\_Asset\\_Files/GPX-Reporter.zip](https://github.com/PacktPublishing/Learning-Geospatial-Analysis-with-Python-Fourth-Edition/raw/main/B19730_11_Asset_Files/GPX-Reporter.zip).

Unzip the `route.gpx` file in the same directory as the `GPX-Reporter.py` script you will create. The `photos` directory, which only has one photo, should be at the same level as well.

Now, we're ready to work through the `GPX-Reporter.py` program. Like other scripts in this book, this program tries to minimize functions so that you can mentally trace the program better and modify it with less effort.

### Importing libraries

First, we will import all the necessary libraries that we'll use throughout the code:

```
from xml.dom import minidom
import urllib.request
import math
import time
import logging
import numpy as np
import srtm
from pygooglechart import SimpleLineChart
from pygooglechart import Axis
from meteostat import Point, Daily, units
from datetime import datetime, timedelta
import fpdf
import glob
from PIL import Image
from PIL import ImageFilter
from PIL import ImageEnhance
```

```
from PIL import ImageDraw
from PIL.ExifTags import TAGS
```

## Setting up logging

Then, we will set up logging to keep track of the program's progress:

```
level = logging.DEBUG
formatter = logging.Formatter("%(asctime)s - %(name)s - %(message)s")
log = logging.getLogger("GPX-Reporter")
log.setLevel(level)
console = logging.StreamHandler()
console.setLevel(level)
console.setFormatter(formatter)
log.addHandler(console)
```

## Helper functions

Then, we will define several helper functions for coordinate conversions, distance calculations, and other utilities.

### *Latitude and longitude to meters*

Our first function converts latitude and longitude to meters, which is required by one of the mapping services we'll use:

```
def ll2m(lat, lon):
    x = lon * 20037508.34 / 180.0
    y = math.log(math.tan((90.0 + lat) * \
        math.pi / 360.0)) / (math.pi / 180.0)
    y = y * 20037508.34 / 180
    return (x, y)
```

### *World coordinates to pixel coordinates*

Our next function converts geospatial coordinates to image coordinates so that we can plot and draw geospatial objects on an image:

```
def world2pixel(x, y, w, h, bbox):
    minx, miny, maxx, maxy = bbox
    xdist = maxx - minx
    ydist = maxy - miny
    xratio = w/xdist
    yratio = h/ydist
    px = w - abs(((maxx - x) * xratio))
```

```

    py = (maxy-y) * yratio
    return int(px), int(py)

```

### *UTC epoch*

Now, we need a function to convert a GPX timestamp to Unix epoch seconds in Greenwich Mean Time, making time math easier:

```

def get_utc_epoch(timestr):
    utctime = time.strptime(timestr, \
        '%Y-%m-%dT%H:%M:%S.000Z')
    secs = int(time.mktime(utctime))
    return secs

```

### *Haversine distance*

Then, we'll use the classic Haversine formula to measure the distance between different points in our GPX track:

```

def haversine(x1, y1, x2, y2):
    x_dist = math.radians(x1 - x2)
    y_dist = math.radians(y1 - y2)
    y1_rad = math.radians(y1)
    y2_rad = math.radians(y2)
    a = math.sin(y_dist/2)**2 + math.sin(x_dist/2)**2 \
        * math.cos(y1_rad) * math.cos(y2_rad)
    c = 2 * math.asin(math.sqrt(a))
    distance = c * (6371/1.609344)
    return distance

```

### *Web mapping service (WMS)*

We will create a function to build a request for the WMS services we'll use for mapping:

```

def wms(minx, miny, maxx, maxy, service, lyr, \
    epsg, style, img, w, h):
    wms = service
    wms += "?SERVICE=WMS&VERSION=1.1.1&REQUEST=GetMap&"
    wms += "LAYERS={}".format(lyr)
    wms += "&STYLES={}&".format(style)
    wms += "SRS=EPSG:{}&".format(epsg)
    wms += "BBOX={}, {}, {}, {}&".format(minx, miny, \
        maxx, maxy)
    wms += "WIDTH={}&".format(w)
    wms += "HEIGHT={}&".format(h)

```

```
wms += "FORMAT=image/jpeg"
wmsmap = urllib.request.urlopen(wms)
with open(img + ".jpg", "wb") as f:
    f.write(wmsmap.read())
```

### ***EXIF tag extraction***

We'll need a function to read the EXIF tags from a photo so that we can eventually get to the GPS data stored there:

```
def exif(img):
    exif_data = {}
    try:
        i = Image.open(img)
        tags = i._getexif()
        for tag, value in tags.items():
            decoded = TAGS.get(tag, tag)
            exif_data[decoded] = value
    except:
        pass
    return exif_data
```

### ***Degrees, minutes, and seconds to decimal degrees***

This function will convert traditional **degrees, minutes, and seconds (DMS)** coordinates to decimal degrees so that we can use them in our other functions. The GPS data in photos is stored in DMS:

```
def dms2dd(d, m, s, i):
    sec = float((m * 60.0) + s)
    dec = float(sec / 3600.0)
    deg = float(d + dec)
    if i.upper() == 'W':
        deg = deg * -1.0
    elif i.upper() == 'S':
        deg = deg * -1.0
    return float(deg)
```

### ***GPS coordinate extraction***

This function will take the complete EXIF data from the photo, extract the GPS coordinates of the photo, and convert them to decimal degrees:

```
def gps(exif):
    lat = None
    lon = None
```

```

if exif['GPSInfo']:
    coords = exif['GPSInfo']
    i = coords[1]
    d = coords[2][0]
    m = coords[2][1]
    s = coords[2][2]
    lat = dms2dd(d, m, s, i)
    i = coords[3]
    d = coords[4][0]
    m = coords[4][1]
    s = coords[4][2]
    lon = dms2dd(d, m, s, i)
return lat, lon

```

## Program variables

We will initialize some program variables that will be used later.

These are our constants to switch back and forth between degrees and radians:

```

deg2rad = 3.141592653589793 / 180.0
rad2deg = 180.0 / 3.141592653589793

```

This is the name of our GPX route file:

```
gpx = "route.gpx"
```

These variables are our WMS parameters:

```

osm_WMS = "http://ows.mundialis.de/services/service"
osm_lyr = "OSM-WMS"
osm_img = "basemap"
osm_epsg = 3857
osm_style = ""

```

These variables are the configuration for our hillshade styling:

```

azimuth = 315.0
altitude = 45.0
z = 5.0
scale = 1.0
no_data = 0

```

These variables set up our elevation image:

```
elv_img = "elevation"
min_clr = (255, 255, 255, 0)
max_clr = (0, 0, 0, 0)
zero_clr = (255, 255, 255, 255)
w = 800
h = 800
```

## Parsing the GPX file

We will parse the GPX file to extract latitude, longitude, elevation, and time data:

```
log.info("Parsing GPX file: {}".format(gpx))
xml = minidom.parse(gpx)
trkpts = xml.getElementsByTagName("trkpt")
lats = []
lons = []
elvs = []
times = []
for trkpt in trkpts:
    lat = float(trkpt.attributes["lat"].value)
    lon = float(trkpt.attributes["lon"].value)
    lats.append(lat)
    lons.append(lon)
    elv = trkpt.getElementsByTagName(\n
        "ele")[0].firstChild.nodeValue
    elv = float(elv)
    elvs.append(elv)
    t = trkpt.getElementsByTagName(\n
        "time")[0].firstChild.nodeValue
    t = get_utc_epoch(t)
    times.append(t)
```

Now, we will find the bounding box of the route:

```
minx = min(lons)
miny = min(lats)
maxx = max(lons)
maxy = max(lats)
```

Then, we will buffer the GPX bounding box by 20% so that the track isn't too close to the edge of the image:

```
xdist = maxx - minx
ydist = maxy - miny
x10 = xdist * .2
y10 = ydist * .2
minx -= x10
miny -= x10
maxx += x10
maxy += x10
```

Then, we will store the bounding box in a single variable to streamline function calls:

```
bbox = [minx, miny, maxx, maxy]
```

We need the bounding box in meters for the OSM WMS service. However, we will download it in degrees to match the elevation file. The WMS spec says the input SRS should match the output, but this custom service just doesn't work that way:

```
mminx, mminy = ll2m(miny, minx)
mmaxx, mmaxy = ll2m(maxy, maxx)
```

## Downloading the basemap and elevation data

We will download the OpenStreetMap basemap and SRTM elevation data:

```
log.info("Downloading basemap")
wms(mminx, mminy, mmaxx, mmaxy, osm_WMS, osm_lyr, \
osm_epsg, osm_style, osm_img, w, h)
log.info("Retrieving SRTM elevation data")
srt = srtm.get_data()
image = srt.get_image((w, h), (miny, maxy), \
(minx, maxx), 300, zero_color=zero_clr, \
min_color=min_clr, max_color=max_clr)
image.save(elv_img + ".png")
```

A basemap is shown in the following screenshot:

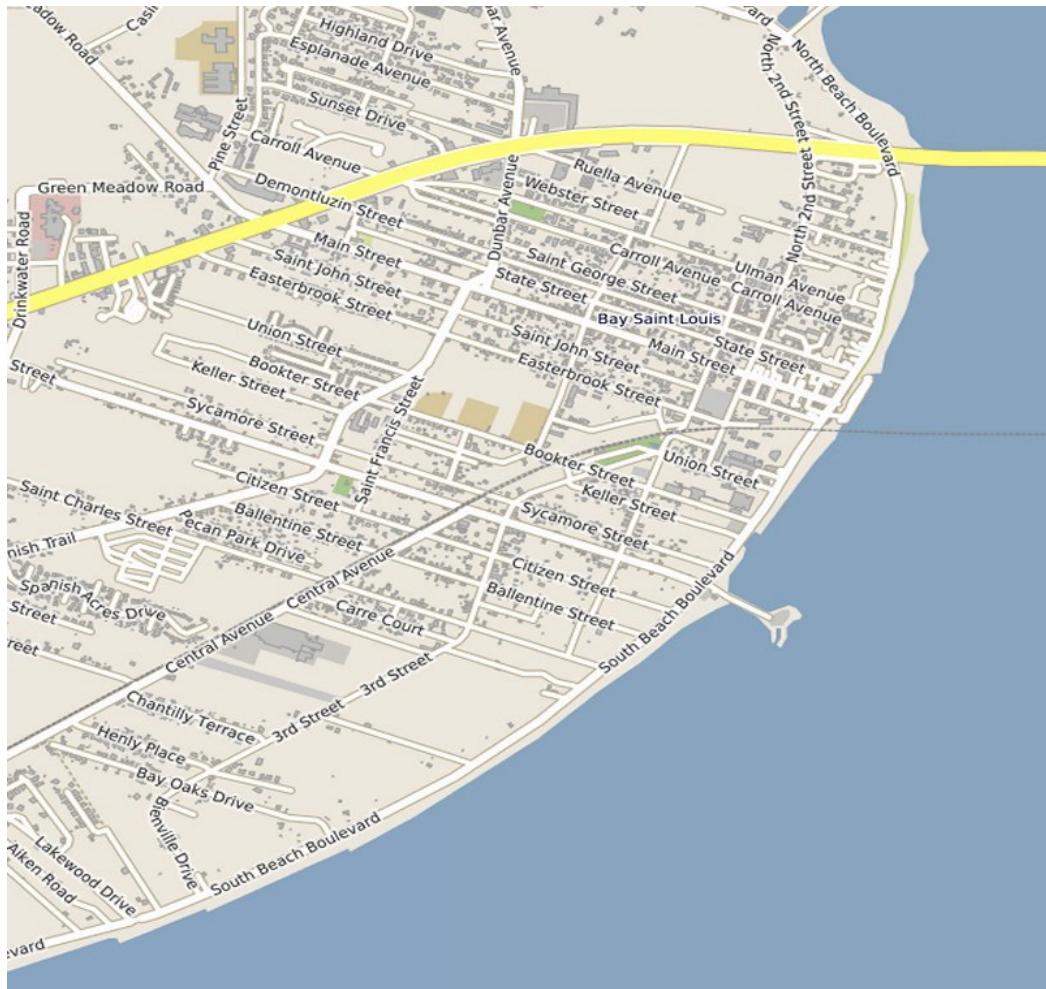


Figure 11.2 – The basemap

Now, let's hillshade the elevation data so that we can see what it looks like.

## Hillshading the elevation data

We will apply hillshading to the elevation data:

```
log.info("Hillshading elevation data")
im = Image.open(elv_img + ".png").convert("L")
dem = np.asarray(im)
window = []
xres = (maxx-minx)/w
yres = (maxy-miny)/h
for row in range(3):
    for col in range(3):
        window.append(dem[row:(row + dem.shape[0]-2), \
                           col:(col + dem.shape[1]-2)])
x = ((z * window[0] + z * window[3] + z * window[3] + z * \
       window[6]) - (z * window[2] + z * window[5] + z * \
       window[5] + z * window[8])) / (8.0 * xres * scale)
y = ((z * window[6] + z * window[7] + z * window[7] + z * \
       window[8]) - (z * window[0] + z * window[1] + z * window[1] + z * \
       window[2])) / (8.0 * yres * scale)
slope = 90.0 - np.arctan(np.sqrt(x*x + y*y)) * rad2deg
aspect = np.arctan2(x, y)
shaded = np.sin(alitude * deg2rad) * np.sin(slope * deg2rad) +
np.cos(alitude * deg2rad) * np.cos(slope * \
deg2rad)* np.cos((azimuth - 90.0) * deg2rad - aspect)
shaded = shaded * 255
relief = Image.fromarray(shaded).convert("L")
for i in range(10):
    relief = relief.filter(ImageFilter.SMOOTH_MORE)
relief.save("relief.png")
```

The hillshaded elevation looks like the following:

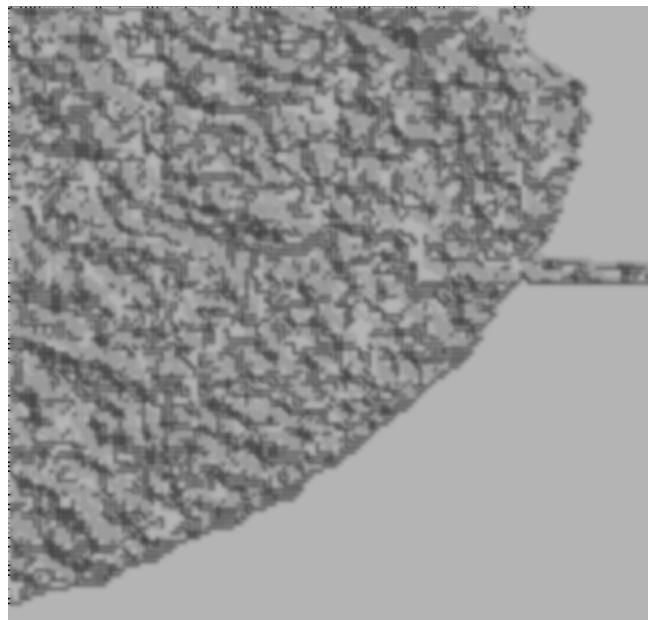


Figure 11.3 – The shaded elevation relief

Now, let's create a map to combine it with.

## Creating a map

We will blend the hillshaded elevation data with the basemap and draw the GPX tracks:

```
log.info("Creating map image")
e = ImageEnhance.Contrast(relief)
relief = e.enhance(2)
base = Image.open(osm_img + ".jpg").crop((0, 0, w-2, h-2))
e = ImageEnhance.Contrast(base)
base = e.enhance(1)
topo = Image.blend(relief.convert("RGB"), base, .9)
points = []

for x, y in zip(lons, lats):
    px, py = world2pixel(x, y, w, h, bbox)
    points.append((px, py))

w -= 2
h -= 2
```

```
track = Image.new('RGBA', (w, h))
track_draw = ImageDraw.Draw(track)
track_draw.line(points, fill=(255, 0, 0, 127), width=4)
topo.paste(track, mask=track)
topo_draw = ImageDraw.Draw(topo)
start_lon, start_lat = (lons[0], lats[0])
start_x, start_y = world2pixel(start_lon, start_lat, w, h, bbox)
start_point = [start_x-10, start_y-10, start_x+10, start_y+10]
topo_draw.ellipse(start_point, fill="lightgreen", outline="black")
start_marker = [start_x-4, start_y-4, start_x+4, start_y+4]
topo_draw.ellipse(start_marker, fill="black", outline="white")

end_lon, end_lat = (lons[-1], lats[-1])
end_x, end_y = world2pixel(end_lon, end_lat, w, h, bbox)
end_point = [end_x-10, end_y-10, end_x+10, end_y+10]
topo_draw.ellipse(end_point, fill="red", outline="black")
end_marker = [end_x-4, end_y-4, end_x+4, end_y+4]
topo_draw.ellipse(end_marker, fill="black", outline="white")
```

## Adding a photo marker

Now, we'll get any photos in the `photo` directory, extract the GPS coordinates, and plot them on the map:

```
images = glob.glob("photos/*.jpg")
for i in images:
    e = exif(i)
    photo_lat, photo_lon = gps(e)
    photo_x, photo_y = world2pixel(photo_lon, photo_lat, w, h, bbox)
    topo_draw.rectangle([photo_x - 12, \
        photo_y - 10, photo_x + 12, photo_y + 10], \
        fill="black", outline="black")
    topo_draw.rectangle([photo_x - 9, photo_y - 8, \
        photo_x + 9, photo_y + 8], fill="white", \
        outline="white")
    topo_draw.polygon([(photo_x-8,photo_y+7), \
        (photo_x-3,photo_y-1), (photo_x+2,photo_y+7)], \
        fill = "black")
    topo_draw.polygon([(photo_x+2,photo_y+7), \
        (photo_x+7,photo_y+3), (photo_x+8,photo_y+7)], \
        fill = "black")
topo.save("{}_topo.jpg".format(osm_img))
```

The blended topographic map with the photo marker looks like the following screenshot:

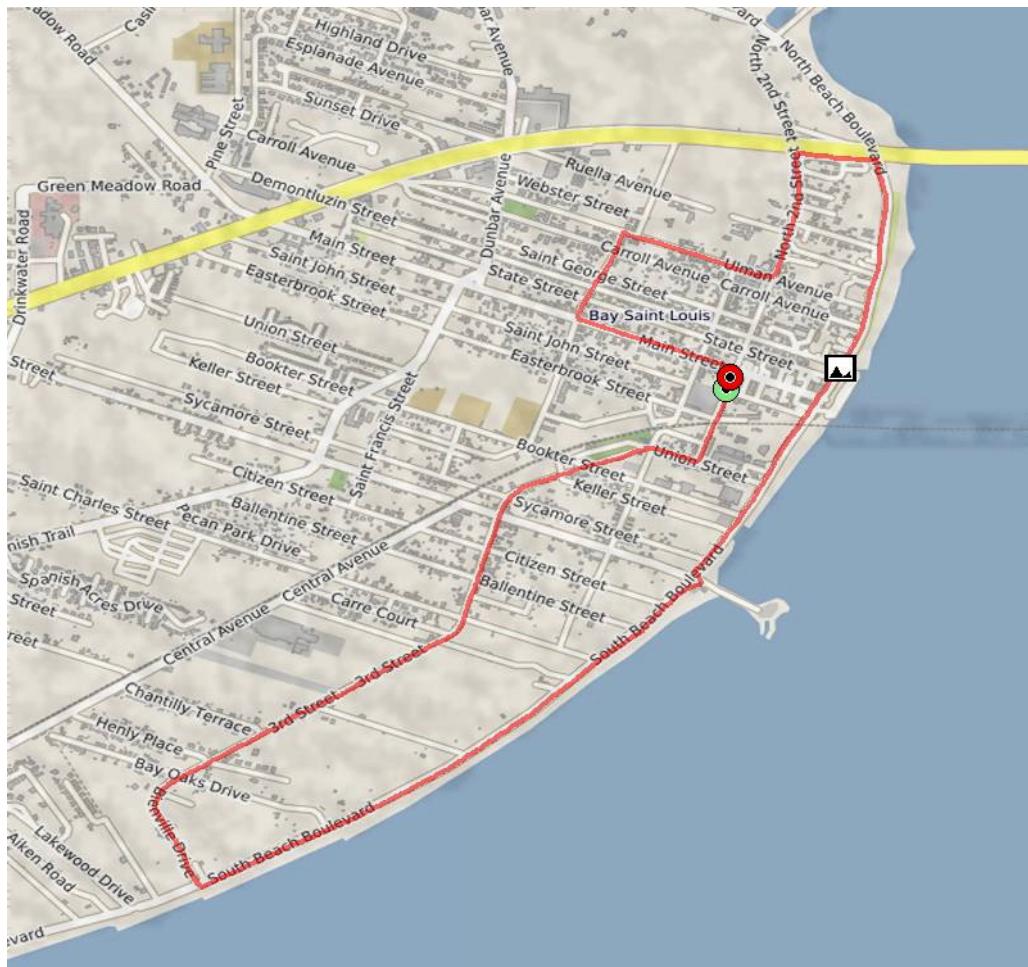


Figure 11.4 – Our completed map with the route, start/end, and photo location

While hillshade mapping gives us an idea of the elevation, it doesn't give us any quantitative data. To get more details, we'll create a simple elevation chart.

## Creating an elevation profile chart

We will create an elevation profile chart using the Google Chart API:

```
log.info("Creating elevation profile chart")
chart = SimpleLineChart(600, 300, y_range=[min(elvs), \
```

```
max(elvs)))
chart.add_data([min(elvs)]*2)
chart.add_data(elvs)
chart.add_data([min(elvs)]*2)

chart.set_colours(['000000'])
chart.add_fill_range('80C65A', 1, 2)
elv_labels = int(round(min(elvs))), \
int(min(elvs)+((max(elvs)-min(elvs))/2))
elv_label = chart.set_axis_labels(Axis.LEFT, elv_labels)
elv_text = chart.set_axis_labels(Axis.LEFT, ["FEET"])
chart.set_axis_positions(elv_text, [30])

distances = []
measurements = []
coords = list(zip(lons, lats))
for i in range(len(coords)-1):
    x1, y1 = coords[i]
    x2, y2 = coords[i+1]
    d = haversine(x1, y1, x2, y2)
    distances.append(d)
total = sum(distances)
distances.append(0)
j = -1

for i in range(1, int(round(total))) :
    mile = 0
    while mile < i:
        j += 1
        mile += distances[j]
    measurements.append((int(mile), j))
    j = -1
positions = []
miles = []
for m, i in measurements:
    pos = ((i*1.0)/len(elvs)) * 100
    positions.append(pos)
    miles.append(m)

miles_label = chart.set_axis_labels(Axis.BOTTOM, miles)
chart.set_axis_positions(miles_label, positions)
miles_text = chart.set_axis_labels(Axis.BOTTOM, \
["MILES", ]) 
```

```
chart.set_axis_positions(miles_text, [50, ])
chart.download('{}_profile.png'.format(elv_img))
```

Our chart should now look like the following screenshot:

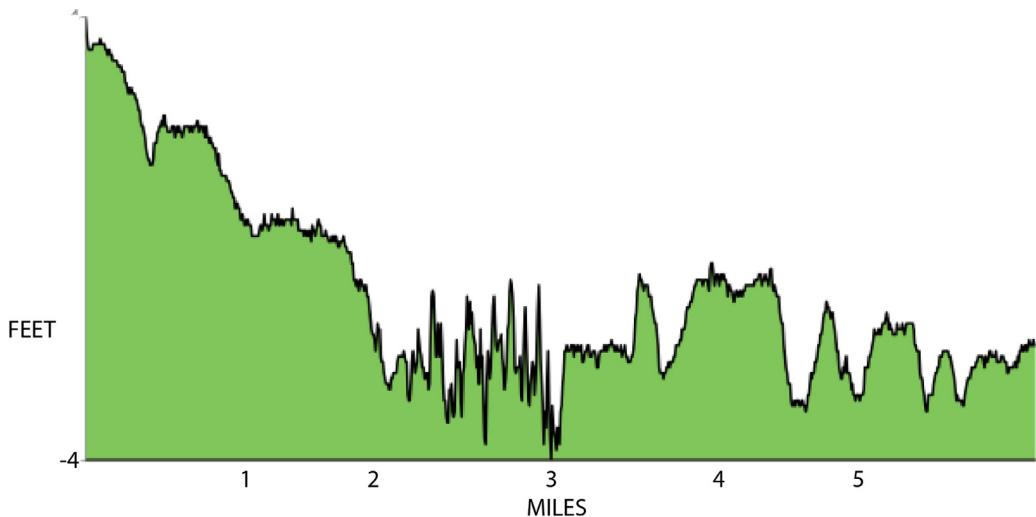


Figure 11.5 – The elevation profile chart

Now, let's create a weather report.

## Creating a weather report

We will fetch weather data for the route and add it to the report:

```
log.info("Creating weather summary")

centx = minx + ((maxx-minx)/2)
centy = miny + ((maxy-miny)/2)
t = times[-1]
p = Point(centy, centx)
weather_start = datetime.fromtimestamp(t)
weather_end = weather_start + timedelta(days=1)
weather_data = Daily(p, weather_start, weather_end)
weather_data = weather_data.convert(units.imperial)
weather_frame = weather_data.fetch()

mintemp = weather_frame['tmin'].iloc[0]
maxtemp = weather_frame['tmax'].iloc[0]
precip = weather_frame['prcp'].iloc[0]
```

## Generating a PDF report

Finally, we will generate a PDF report that includes the map, elevation profile, and weather summary:

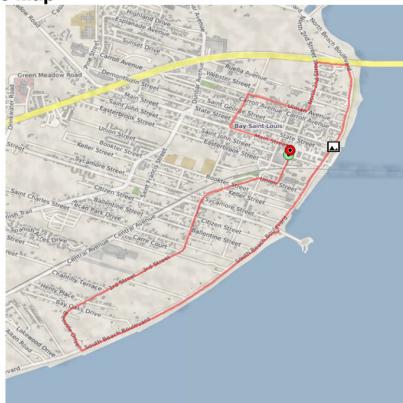
```
pdf = fpdf.FPDF("P", "in", "Letter")
pdf.add_page()
pdf.set_font('Arial', 'B', 20)
pdf.cell(6.25, 1, 'GPX Report', border=0, align="C")
pdf.ln(h=1)
pdf.cell(1.75)

pdf.cell(4, border="T")
pdf.ln(h=0)
pdf.set_font('Arial', style='B', size=14)
pdf.cell(w=1.2, h=1, txt="Route Map", border=0, align="C")
pdf.image("{}_topo.jpg".format(osm_img), 1, 2, 4, 4)
pdf.ln(h=4.35)
pdf.set_font('Arial', style='B', size=14)
pdf.cell(w=1.2, h=1, txt="Elevation Profile", border=0, align="C")
pdf.image("{}_profile.png".format(elv_img), 1, 6.5, 4, 2)
pdf.ln(h=2.4)
pdf.set_font('Arial', style='B', size=14)
pdf.cell(1.2, 1, "Weather Summary", align="C")
pdf.ln(h=.25)
pdf.set_font('Arial', style='', size=12)
pdf.cell(1.8, 1, "Min. Temp.: {}".format(mintemp), align="L")
pdf.ln(h=.25)
pdf.cell(1.8, 1, "Max. Temp.: {}".format(maxtemp), align="L")
pdf.cell(1.2, 1, "Precip.: {}".format(precip), align="L")
pdf.ln(h=.25)
pdf.ln(h=2.4)
pdf.set_font('Arial', style='B', size=14)
pdf.cell(1.2, 1, "Photos", align="C")
pdf.ln(h=.25)
for i in images:
    pdf.image(i, 1.2, 1, 3, 3)
    pdf.ln(h=.25)
log.info("Saving report pdf")
pdf.output('gpx_report.pdf', 'F')
```

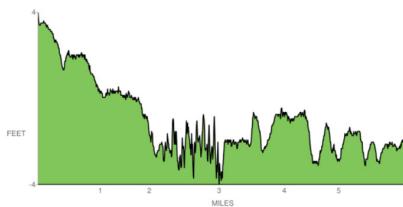
You should have a PDF document in your working directory called `gpx-report.pdf` that contains your finished product. It should look like the following screenshot:

## GPX Report

### Route Map



### Elevation Profile



### Weather Summary

Min. Temp.: 55.0

Max. Temp.: 71.1

Precip.: 0.0

### Photos



Figure 11.6 – The final GPX report

With this report, we have used all the techniques we've learned throughout this book and built a GPS reporting tool!

## Summary

Congratulations! You created a complete report for a running route, similar to reports created by some of the leading commercial exercise apps! You combined terrain data, street mapping data, weather data, tracking data, geolocated photos, and charts into a single, comprehensive report. This type of synthesis is a perfect example of the extent to which you can turn raw geospatial data sources into truly useful information that can help shape future decisions. This kind of information product is the promise of geospatial analysis and the goal of this book was to teach you how to create this.

In this book, you pulled together the most essential tools and skills needed to be a modern geospatial analyst. Whether you use geospatial data occasionally or use it all the time, you will be better equipped to make the most of geospatial analysis. This book focused on using open source tools almost entirely found within the PyPI directory, for ease of installation and integration. However, even if you use Python as a driver for a commercial GIS package or a popular library such as GDAL, the ability to test out new concepts in pure Python will always come in handy.

## Questions

1. Which Python module allows us to log program status to multiple output streams?
2. What are the informational data tags embedded in some images called?
3. Which Python module lets us manipulate date information?

## Further reading

- Python provides a rich set of libraries for visualizing data. One of the most prominent is `matplotlib`, which can produce numerous types of charts and maps and save them to PDF. Packt has several books on Matplotlib that can take you even further in this direction, including *Matplotlib 3.0 Cookbook*: <https://www.packtpub.com/product/matplotlib-30-cookbook/9781789135718>.



# Assessments

## *Chapter 1 – Learning about Geospatial Analysis with Python*

1. The Canadian Department of Forestry and Rural Development
2. Thematic map
3. The capability to display geographic data and the capability to analyze it

## *Chapter 2 – Learning about Geospatial Data*

1. Vector and raster data
2. Spatial indexing algorithms
3.  $x, y, z$ :  $x$  is the east-west axis,  $y$  is the north-south axis, and  $z$  is the height or elevation

## *Chapter 3 – The Geospatial Technology Landscape*

1. Data access, computational geometry, image processing, desktop tools (and other visualization tools), and metadata management.
2. If the software is built on existing libraries, what value does it add? Why did the developers create it?

## *Chapter 4 – Geospatial Python Toolbox*

1. Anaconda and pip
2. numpy
3. MySQL, PostgreSQL (PostGIS), and SQLite (Spatialite)

## *Chapter 5 – Python and Geospatial Algorithms*

1. The Vincenty formula
2. Reprojection
3. A point

## ***Chapter 6 – Creating and Editing GIS Data***

1. Aggregation
2. Dot density, choropleth, heat map
3. Geocoding

## ***Chapter 7 – Python and Remote Sensing***

1. Histogram
2. Classification
3. Extraction

## ***Chapter 8 – Python and Elevation Data***

1. ASCIIGRID
2. Slope and aspect
3. Triangulated irregular network or TIN
4. Bathymetry

## ***Chapter 9 – Advanced Geospatial Modeling***

1. NDVI = (Infrared – Red) / (Infrared + Red)
2. Flood inundation model
3. Djikstra and A-star (A\*)

## ***Chapter 10 – Working with Real-Time Data***

1. Data currency
2. General Transit Feed Specification
3. Web mashup

## ***Chapter 11 – Putting It All Together***

1. The Logging module
2. EXIF tags
3. The Datetime module

# Index

## Symbols

**3D geospatial data models** 74  
disaster management 74  
environmental management 74  
urban planning and design 74

## A

**A\*** algorithm 333  
reference link 333  
**Adobe Photoshop** 324  
**Advanced Spaceborne Thermal Emission  
and Reflection Radiometer (ASTER)** 18  
**aerial images**  
used, for colorizing LiDAR data 298-302  
**aggregation** 209  
**algorithms**  
reference link 178  
**ambitious OpenStreetMap project**  
reference link 22  
**Anaconda** 122, 123  
Miniconda 123  
**ArcGIS** 110  
reference link 110  
**ArcGIS Network Analyst**  
reference link 102

**ArcSDE** 98  
**artificial intelligence (AI)** 114  
**ASCIIIGRID Digital Elevation  
Model (DEM)** 273  
**ASCII grid files** 69, 70  
accessing 270  
grids, reading 270-272  
grids, writing 272-274  
**ASCII text files** 68  
**aspect grid**  
creating 275  
**Asynchronous JavaScript and  
XML (AJAX)** 51  
**Atlas of Living Australia (ALA)** 20  
**attribute selections**  
performing 207, 208  
**Automated Identification System (AIS)** 226  
**automated remote sensing (ARS)** 250

## B

**band math** 36  
**bathymetry**  
working with 304-307  
**binary large objects (blobs)** 24, 68  
**box selections**  
bounding 206, 207

**buffer operation** 31  
**Bureau of Ocean Energy, Management, Regulation, and Enforcement (BOEMRE)** 98

## C

**California Cooperative Oceanic Fisheries Investigations (CalCOFI)** 88  
**Canada Geographic Information System (CGIS)** 10  
**Canadian Hydrographic Service (CHS)** 97  
**Catalog Service for the Web (CSW)** 25, 112  
**Centre National d'études Spatiales (CNES)** 17  
**change detection** 36, 259-264  
**ChatGPT** 114, 183  
    reference link 115  
    used, for extracting image footprints 264-267  
    used, for measuring polygon perimeter 183-187  
**choropleth maps** 220-223  
**classification** 251  
**color hillshade**  
    creating 330-332  
**Comma-Separated Value (CSV) file** 50, 127  
**Common Language Runtime (CLR)** 120  
**common raster data concepts** 35  
    band math 36  
    change detection 36  
    feature extraction 37  
    histogram 37  
    supervised classification 38  
    unsupervised classification 38  
**common vector GIS concepts** 28  
    buffer operation 31  
    data structure 28-30

dissolve operation 31  
generalize operation 32  
intersection operation 32  
join operation 35  
merge operation 33  
point in polygon 34  
union operation 34, 35  
**compressed formats** 69  
**computational geometry** 87  
    Computational Geometry Algorithms Library (CGAL) 89  
    Esri Network Analyst 101  
    Esri Spatial Analyst 101  
    Geometry Engine - Open Source (GEOS) 91  
    geospatial databases 96  
    Java Topology Suite (JTS) 90  
    PostGIS 92-95  
    PROJ projection library 88, 89  
    routing 101  
**Computational Geometry Algorithms Library (CGAL)** 89  
    reference link 90  
**computer-aided design (CAD)** 63  
**computer-aided drafting (CAD)** 19  
**computer programming** 19  
**Conda** 122  
**Content Standard for Digital Geospatial Metadata (CSDGM)** 58  
**coordinate conversion** 176, 178  
**coordinate format conversion** 181, 182

## D

**data**  
    acquiring, with Python  
    networking libraries 125  
**data access** 82, 84  
    Point Data Abstraction Library (PDAL) 87

**database management system (DBMS)** 24  
**data currency** 357  
**data formats**  
  overview 50-53  
**data sharing**  
  with interchange formats 76, 77  
**data structure** 28-30, 53  
  common traits 54  
**datum** 168  
**degrees decimal minutes** 227  
**degrees, minutes, and seconds (DMS)** 378  
**Department of the Interior (DOI)** 15  
**desktop tools** 103  
  ArcGIS 110  
  Google Earth 106-108  
  GRASS GIS 104, 105  
  gvSIG 105  
  Leaflet 111  
  NASA WorldWind 108, 109  
  OpenJUMP 106  
  OpenLayers 111  
  Quantum GIS 103, 104  
**Digital Elevation Model (DEM)** 17, 323  
**Dijkstra algorithm** 333  
  reference link 333  
**dissolve operation** 31  
**distance**  
  measuring 166-169  
  measuring, with haversine formula 171-173  
  measuring, with Pythagorean theorem 169-171  
  measuring, with Vincenty formula 173-75  
**Doppler Weather Radar (DWR)** 358  
**dots per inch (dpi)** 32  
**Drawing (DWG)** 63  
**Drawing Exchange Format (DXF)** 63

**E**

**Earth Observation Satellite (EOSAT)** 15  
**Earth Resources Technology Satellite (ERTS)** 15  
**ElementTree module** 136, 137  
  used, for building XML 138-140  
**elevation contours**  
  creating 279-283  
**elevation formats**  
  using 270  
**Enhanced Compression Wavelet (ECW)** 69  
**ENVI** 82  
**ERDAS IMAGINE** 82  
**Esri File Geodatabase** 100  
  reference link 100  
**Esri Network Analyst** 101  
**Esri shapefile**  
  reference link 61  
**Esri Spatial Analyst** 101  
**Euclidean distance**  
  reference link 171  
**Eurasian Plate** 168  
**European Petroleum Survey Group (EPSG)** 25, 52  
**European Terrestrial Reference System 1989 (ETRS89)** 168  
**EXIF tags** 348

**F**

**false color images** 28  
**feature extraction** 37  
**Feature Manipulation Engine (FME)** 61  
**field data collection form** 368  
  working 369-371  
**field descriptors** 194

- files**  
building 131-133  
compressing 131-133
- file structure** 58-60
- File Transfer Protocol (FTP)** 125, 129-131
- Fiona library** 149, 150
- flood fill algorithm** 324  
reference link 325
- flood fill function** 325-330
- flood inundation model**  
creating 323-325
- Folium library** 161, 162
- G**
- GDAL library** 151
- generalize operation** 32
- General Transit Feed Specification (GTFS) feed** 359
- geocoding**  
using, for turning addresses into points 227-229
- geodatabases** 24
- geodetic system** 168
- Geographic Information System (GIS)** 3, 165  
evolution 10, 11
- Geographic Markup Language (GML)** 64
- Geographic Resources Analysis Support System (GRASS)** 11, 82
- Geography Markup Language (GML)** 106
- geoid** 52
- GeoJSON** 66, 67
- geojson module** 146
- Geometry Engine - Open Source (GEOS)** 84, 91, 92, 141  
reference link 92
- GeoNetwork** 113  
reference link 114
- GeoNode** 112  
reference link 112
- GeoPackage** 67, 101  
reference link 101
- GeoPandas library** 156, 157
- geospatial analysis** 4-6, 19  
for object-oriented programming 20, 21  
history 6-9  
importance 21, 22
- Geospatial Data Abstraction Library (GDAL)** 19, 68, 83, 84  
raster data 84  
vector data 85, 86
- geospatial database** 76, 96  
ArcSDE 98  
Esri File Geodatabase 100  
GeoPackage 101  
Microsoft SQL Server 99  
MySQL 99  
Oracle Spatial and Graph 96, 97  
SpatiaLite 100
- geospatial model** 311
- GeoTools** 84
- GIS analysis**  
performing, with multiprocessing 229, 230
- GIS concepts** 22  
map projections 25, 26  
metadata 24, 25  
rendering 26, 27  
spatial databases 24  
spatial indexing 24  
thematic map 22-24
- Global Positioning System (GPS)** 52
- Google Earth** 106-108  
reference link 108

**Google Mercator** 52

**GPS data**

using 226, 227

**GPS report** 374, 375

**GPS report tool**

basemap and elevation

data, downloading 381, 382

building 375

elevation data, hillshading 383, 384

elevation profile chart, creating 386, 388

GPX file, parsing 380, 381

helper function 376

libraries, importing 375

logs, setting up 376

map, creating 384

PDF report, generating 389, 391

photo marker, adding 385, 386

program variables 379

weather report, creating 388

**graphical user interface (GUI)** 11

**GRASS GIS** 104, 105

reference link 105

**GRIB** 68

**grids** 57

creating, from Light Detection

and Ranging (LiDAR) data 284-290

reading 270-272

writing 272-274

**gvSIG Community Edition (gvSIG CE)** 105

reference link 106

## H

**haversine formula**

used, for measuring distance 171-173

**HDF5** 68

**heat maps**

creating 225, 226

**helper function, GPS report**

degrees, minutes, and seconds (DMS) 378

EXIF tag extraction 378

GPS coordinate extraction 378

Haversine distance 377

latitude and longitude to meters 376

UTC epoch 377

web mapping service (WMS) 377

world coordinates to pixel coordinates 376

**histogram** 37

**Hue, Saturation, and Value (HSV)** 292

## I

**image bands**

swapping 236-239

**image footprints**

extracting, with ChatGPT 264-267

**image histograms**

creating 239-243

stretch, performing 243-245

**images**

classifying 250-253

clipping 245-250

features, extracting from 253-259

**images, creating for visualization** 217

choropleth maps 220-223

dot density calculations 217-220

heat maps, creating 225, 226

spreadsheets, using 223-225

**Infrastructure for Spatial Information**

in European Community (INSPIRE) 58

**interchange formats**

used, for data sharing 76, 77

**International Association of Oil**

and Gas Producers (IOGP) 25

**International Geophysical Year (IGY)** 13

**intersection operation** 32

## J

- JavaScript Object Notation (JSON)** 66, 143
- Java Topology Suite (JTS)** 84, 90
  - reference link 91
- Java Unified Mapping Platform (JUMP)** 106
- join operation** 35
- json module** 144-146
- Jupyter** 123
  - PyPI 124
  - Python Installation Program (pip) 124
  - Python virtualenv module 124

## K

- Keyhole Markup Language (KML)** 21, 64, 107

## L

- Landsat Data Continuity Mission (LDCM)** 16
- large language models (LLMs)** 114, 183
- LAS** 284
- Leaflet** 111
- least cost path analysis**
  - performing 333-336
- Lempel-Ziv-Welch (LZW)** 69
- LIDAR Exchange Format (LAS)** 74
- Light Detection and Ranging (LiDAR)** 73, 74, 97, 233
- Light Detection and Ranging (LiDAR) data**
  - classifying 302-304
  - colorizing, with aerial images 298-301
  - grid, creating from 284-290
  - triangulated irregular network (TIN), creating 295-297

visualizing, with Python Imaging Library (PIL) 290-294

working with 283, 284

### line direction

calculating 176

### look-up table (LUT)

lossless methods 27

lossy compression algorithms 27

## M

- Manhattan distance** 336
- map projections** 25, 26
- marching squares**
  - reference link 279
- markup-based formats** 64, 65
- Mercator** 51
  - reference link 52
- merge operation** 33
- metadata** 24-58
- metadata management** 112
  - GeoNetwork 113
  - GeoNode 112
  - Python's pycsw library 112
- Microsoft SQL Server** 99
  - reference link 99
- Minesweeper** 324
- Miniconda** 123
- minidom module** 134, 135
  - used, for building XML 138-140
- Mississippi Transverse Mercator (MSTM)** 169
- M shape type** 198
- Multi-resolution Seamless Image Database (MrSID)** 69
- Multispectral Scanner (MSS)** 15
- MySQL** 99
  - reference link 100

## N

- NASA WorldWind** 108, 109
  - reference link 110
- National Aeronautics and Space Administration (NASA)** 108
- National Marine Electronics Association (NMEA)** 226
- National Oceanic and Atmospheric Administration (NOAA)** 15, 88, 363
- Network Common Data Form (NetCDF)** 68
- normalized difference vegetation index (NDVI)** 28, 312
  - calculating, with formula 318, 319
  - classes, creating 321
  - classifying 319
  - creating 312-314
  - data, loading 315, 316
  - framework setup 314, 315
  - functions 319
  - image, classifying 321
  - loading 320
  - output, saving 322
  - preparing 320
  - shapefile, rasterizing 316, 317
- NumPy library** 151, 152

## O

- Oak Ridge National Laboratory (ORNL)** 36
- object-oriented programming**
  - for geospatial analysis 20, 21
- Ocean Observing System (PacIOOS)** 112
- octree data structure** 54
- OGC OpenAPI**
  - reference link 75
- OGR library** 147

**Open Geospatial Consortium (OGC)** 25, 50, 86

- OpenJUMP**
  - reference link 106
- OpenLayers** 111
- Open Source Geospatial Foundation (OSGeo)** 88
- Open Source Software Image Map (OSSIM)** 82
- OpenStreetMap (OSM)** 64, 161, 228
  - reference link 26
- Oracle Spatial and Graph** 96, 97
- Oracle Spatial Database Option (SDO)** 97
- Orfeo ToolBox (OTB)** 82
- OSMnx library** 161
- overview data** 57, 58

## P

- pgRouting** 102
  - reference link 102
- photos**
  - geolocating 348-351
- PIL library** 153, 154
- PNGCanvas library** 154, 155
- point cloud** 283
- point cloud data** 16-18, 73
  - Light Detection and Ranging (LiDAR) 73, 74
- Point Data Abstraction Library (PDAL)** 87
- point in polygon** 34
- point-in-polygon formula** 204, 205
- points** 198
- polygon** 30
  - area, calculating 182, 183
- polygon holes**
  - ignoring 259

- polygon perimeter**  
measuring, with ChatGPT 183-187
- PostGIS** 92-95  
features 95  
reference link 95
- PROJ library** 84
- PROJ projection library** 88, 89  
reference link 89
- PyFPDF library** 157
- PyMySQL library** 158-160
- Pypi** 124
- pyramids** 57
- PyShp library** 148
- Pythagorean theorem**  
used, for measuring distance 169-171
- Python** 38, 39
- Python Imaging Library**  
(PIL) 152, 221, 233, 313, 364  
using, to visualize LiDAR data 290-294
- Python Installation Program (pip)** 124
- Python JSON libraries** 143  
geojson module 146  
json module 144-146
- Python markup** 133  
ElementTree module 136, 137  
minidom module 134, 135  
XML, building with ElementTree  
module 137-140  
XML, building with minidom  
module 137-140
- Python networking libraries**  
File Transfer Protocol (FTP) 129-131  
for acquiring data 125  
Python requests module 128, 129  
Python urllib module 126-128
- Python requests module** 128, 129  
features 128
- Python's pycsw library** 112
- Python urllib module** 126-128
- Python virtualenv module** 124  
exploring 124, 125
- Q**
- quadtree index** 54, 55
- quality assurance (QA) metadata** 352
- Quantum GIS (QGIS)** 11, 103, 104, 246  
reference link 102  
using 121
- R**
- radians**  
reference link 171
- Radio Corporation of America (RCA)** 15
- raster data** 53, 67  
properties, examining 234-236
- raster data, types** 68  
ASCII grid files 69, 70  
BMP format 69  
compressed formats 69  
GIF format 69  
JPEG format 69  
PNG format 69  
TIFF files 68  
word files 70-72
- rasterio library** 160
- raster path**  
converting, to shapefile 343, 344
- ray-casting method** 204
- real terrain example** 336-338  
A\* algorithm, implementing 339-341  
grid, loading 338  
helper functions, defining 339  
path, creating 341-343

**real-time data** 358

limitations 358

using 358

**Red, Green, and Blue (RGB)** 28, 236**relational database management system (RDBMS)** 24**remote sensing** 11-15, 27

images as data 27

remote sensing and color 28

**rendering** 26**reports**

gathering, from field 368-371

**REpresentational State Transfer (REST)** 75**reprojection** 178-181**Return Beam Vidicon (RBV)** 15**rio-cloudmask**

reference link 355

**routing** 101

along streets 344-347

pgRouting 102, 103

**R-tree index** 54-56**Runkeeper**

reference link 374

**S****satellite image cloud cover**

calculating 351-355

**Scalable Vector Graphics (SVG)** 65**shaded relief**

creating 274-279

**shapefiles**

accessing 191-193

attributes, reading 193-197

changing 198, 200

editing 190, 191

fields, adding 201

geometry, aggregating 209-211

geometry, extracting 211, 212

geometry, reading 198

merging 201, 202

point-in-polygon formula 204, 205

polygon faces, connecting

to nearest point 212-216

raster path, converting 343, 344

selections, performing 204

splitting 203

subsetting spatially 203, 204

**Shapely library** 149**Shuttle Radar Topography****Mission (SRTM)** 17**Shuttle Radar Topography Mission****(SRTM) dataset** 323

reference link 323

**SimpleGIS** 38

building 39, 40

creating 38

data model, setting up 40-42

map, rendering 43-47

**slippy maps** 51**slope grid**

creating 275

**software development kit (SDK)** 109**sound navigation and ranging (sonar)** 305**Space Imaging** 15**spatial databases** 24**spatial data engine (SDE)** 98**spatial indexing** 24, 54**spatial indexing algorithms** 54

grids 57

quadtree index 55

R-tree index 55, 56

**SpatiaLite** 100

reference link 100

**Spatial Reference**

reference link 66

**spatiotemporal data** 77, 78

**spreadsheets**

using 223-225

**string formatting** 137

**Structured Query Language (SQL)** 24

**supervised classification** 38, 251

## T

**tab-separated values (TSV)** 50

**tag-based formats** 64, 65

**tag-based parsers** 133

**Tagged Image File Format (TIFF)** 68

**TerraServer** 50

**thematic map** 22-24

**third-party Python modules**

Anaconda 122, 123

installing 122

**threshold-based classification** 302

**Tiled Maps** 51

**Topologically Integrated Geographic Encoding and Referencing (TIGER)** 95

**Traveling Salesman Problem (TSP)** 101

reference link 101

**triangulated irregular network (TIN)** 74, 97

creating 295-297

## U

**Uniform Resource Locator (URL)** 125

**union operation** 34, 35

**United States Geological Survey (USGS)** 127

**unsupervised classification** 38, 251

**US Federal Geographic Data**

Committee (FGDC) 58

**US Geological Survey (USGS)** 16, 88

**US National Oceanic and Atmospheric**

Administration (NOAA) 129

## V

**vector data** 53, 60

**vector data, types** 60

computer-aided design (CAD) 63

GeoJSON 66, 67

GeoPackage 67

markup-based formats 65

shapefiles 61, 63

tag-based and markup-based formats 64

tag-based formats 65

**vector footprints** 264

**vehicles, tracking**

vehicle location, mapping 361-363

vehicle location, obtaining 359, 360

weather data, adding on map 363-368

WMATA real-time GTFS feed, using 359

**Vincenty formula**

reference link 175

used, for measuring distance 173-175

**virtual environments** 122

## W

**Web Feature Service (WFS)** 64, 75, 106

**Web Map Server (WMS)** 106

**Web Map Service (WMS)** 50, 75

**web mashups**

real-time data, using 358

reference link 359

**Web Mercator** 26

**web services** 75, 76

**Well-Known Binary (WKB)** 76, 141

**Well-Known Text**

(WKT) 25, 64, 119, 141, 142

**Well-Known Text (WKT) format** 133

**WGS84** 168

**WMATA real-time GTFS feed** 359

**word files** 70-72

**WorldDEM** 18

**World Health Organization (WHO)** 4

## X

**XML**

building, with ElementTree module 137-140

building, with minidom module 137-140

**XPath**

reference link 136

## Z

**z-values** 198





[www.packtpub.com](http://www.packtpub.com)

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

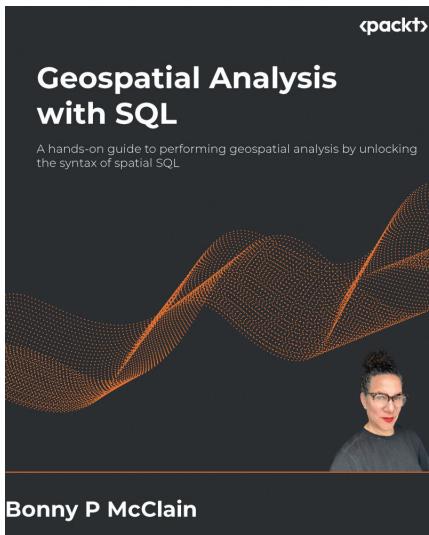
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packtpub.com](http://packtpub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub.com](mailto:customercare@packtpub.com) for more details.

At [www.packtpub.com](http://www.packtpub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

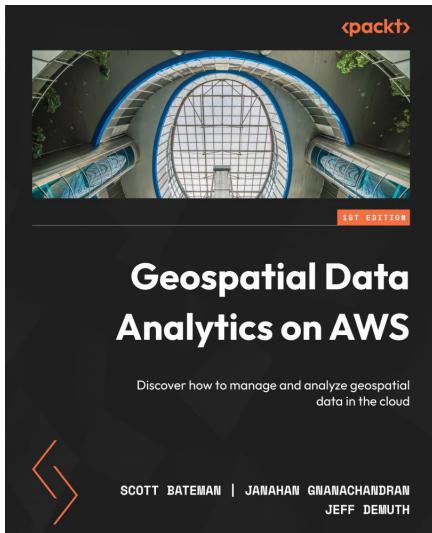


## Geospatial Analysis with SQL

Bonny P McClain

ISBN: 978-1-80461-646-8

- Understand geospatial fundamentals as a basis for learning spatial SQL
- Generate point, line, and polygon data with SQL
- Use spatial data types to abstract and encapsulate spatial structures
- Work with open source GIS combined with plug-ins
- Visualize spatial data and expand QGIS functionality with Postgres
- Apply location data to leverage spatial analytics
- Perform single-layer and multiple-layer spatial analyses



## Geospatial Data Analytics on AWS

Scott Bateman, Janahan Gnanachandran

ISBN: 978-1-80461-382-5

- Discover how to optimize the cloud to store your geospatial data
- Explore management strategies for your data repository using AWS Single Sign-On and IAM
- Create effective SQL queries against your geospatial data using Athena
- Validate postal addresses using Amazon Location services
- Process structured and unstructured geospatial data efficiently using R
- Use Amazon SageMaker to enable machine learning features in your application
- Explore the free and subscription satellite imagery data available for use in your GIS

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share your thoughts

Now you've finished *Cloud Penetration Testing for Red Teamers*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

---

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781837639175>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly