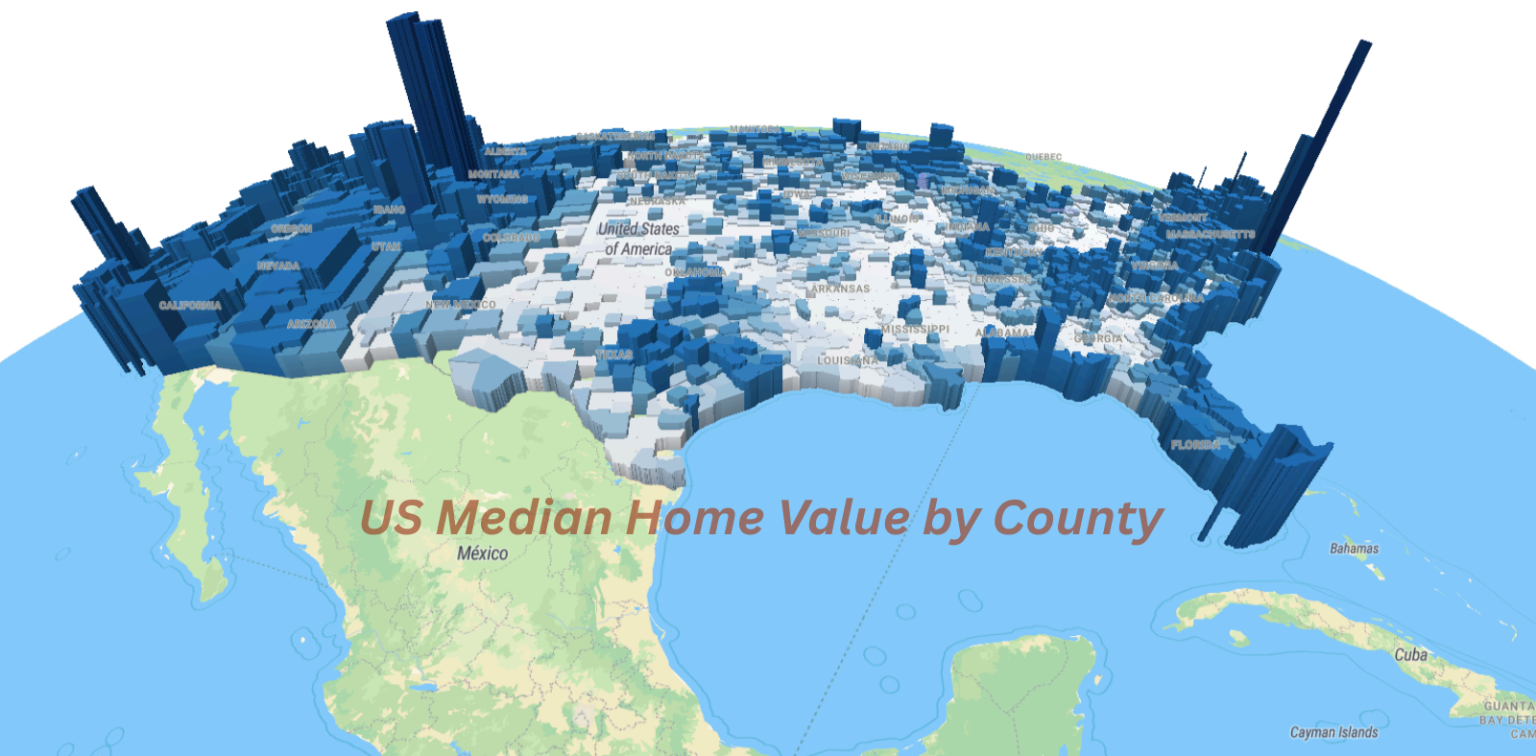


Introduction to GIS Programming

A Practical Python Guide to
Open Source Geospatial Tools

Qiusheng Wu



Introduction to GIS Programming

A Practical Python Guide to Open Source
Geospatial Tools

Qiusheng Wu
2025

Contents

Preface	1
Introduction	3
Who This Book Is For	3
What This Book Covers	4
Getting the Most Out of This Book	5
Conventions Used in This Book	5
Downloading the Code Examples	6
Video Tutorials	6
Get in Touch	7
Acknowledgments	7
About the Author	8
Licensing and Copyright	8
I: Software Setup	9
1. Overview of Software Tools	11
1.1. Introduction	11
1.2. Learning Objectives	11
1.3. Essential Software Tools	11
1.4. Tool Integration and Workflow	13
1.5. Running Code Examples	14
1.6. Key Takeaways	14
2. Introduction to Python Package Management	15
2.1. Introduction	15
2.2. Learning Objectives	15
2.3. Installing Conda (Miniconda)	16
2.4. Understanding Conda Concepts	17
2.5. Creating Your First Geospatial Environment	18
2.6. Troubleshooting Conda	19
2.7. Essential Conda Commands	20
2.8. Introducing uv: The Fast Alternative	24
2.9. Best Practices for Package Management	25
2.10. Key Takeaways	26
2.11. Exercises	26
3. Setting Up Visual Studio Code	28
3.1. Introduction	28
3.2. Learning Objectives	28
3.3. Installing Visual Studio Code	29
3.4. Essential Extensions for Python Programming	29
3.5. Configure VS Code for Python Development	31
3.6. Essential Keyboard Shortcuts	32
3.7. References and Further Learning	34
3.8. Key Takeaways	34
3.9. Exercises	34
4. Version Control with Git	36
4.1. Introduction	36
4.2. Learning Objectives	37
4.3. Setting Up GitHub Account	37

4.4.	Installing Git	37
4.5.	Configuring Git	38
4.6.	Understanding Git Concepts	38
4.7.	Essential Git Commands	39
4.8.	Using GitHub	42
4.9.	Integration with VS Code	43
4.10.	Best Practices for Geospatial Projects	43
4.11.	Key Takeaways	44
4.12.	Exercises	45
5.	<i>Using Google Colab</i>	47
5.1.	Introduction	47
5.2.	Learning Objectives	47
5.3.	Getting Started with Google Colab	47
5.4.	Setting Up Your Geospatial Environment	48
5.5.	Essential Colab Features	50
5.6.	Run Code Examples in Colab	51
5.7.	Key Takeaways	51
5.8.	Exercises	52
6.	<i>Working with JupyterLab</i>	53
6.1.	Introduction	53
6.2.	Learning Objectives	53
6.3.	Installing and Setting Up JupyterLab	53
6.4.	Getting Started with JupyterLab	54
6.5.	Essential Keyboard Shortcuts	57
6.6.	Running Code Examples on MyBinder	59
6.7.	Key Takeaways	59
6.8.	Exercises	59
7.	<i>Using Docker</i>	62
7.1.	Introduction	62
7.2.	Learning Objectives	62
7.3.	Installing Docker Desktop	62
7.4.	Basic Concepts	64
7.5.	Running Code Examples in Docker	64
7.6.	Common Docker Commands	65
7.7.	Key Takeaways	66
7.8.	Exercises	67
II:	Python Programming Fundamentals	69
8.	<i>Variables and Data Types</i>	71
8.1.	Introduction	71
8.2.	Learning Objectives	71
8.3.	Variables in Python	71
8.4.	Naming Variables	72
8.5.	Data Types	73
8.6.	Escape Characters	74
8.7.	Comments in Python	74
8.8.	Working with Variables and Data Types	74
8.9.	Basic String Operations	75
8.10.	Key Takeaways	76

8.11. Exercises	76
9. Python Data Structures	78
9.1. Introduction	78
9.2. Learning Objectives	78
9.3. Tuples	78
9.4. Lists	79
9.5. Sets	81
9.6. Dictionaries	84
9.7. Data Structure Selection Guide	87
9.8. Key Takeaways	87
9.9. Exercises	88
10. String Operations	90
10.1. Introduction	90
10.2. Learning Objectives	90
10.3. Creating and Manipulating Strings	90
10.4. String Methods for Geospatial Data	92
10.5. String Formatting	95
10.6. String Operation Decision Guide	98
10.7. Key Takeaways	98
10.8. Exercises	99
11. Loops and Conditional Statements	101
11.1. Introduction	101
11.2. Learning Objectives	101
11.3. For Loops	101
11.4. While Loops	103
11.5. Control Statements: Making Decisions in Your Code	104
11.6. Combining Loops and Control Statements	105
11.7. Loop and Control Statement Decision Guide	107
11.8. Key Takeaways	108
11.9. Exercises	108
12. Functions and Classes	110
12.1. Introduction	110
12.2. Learning Objectives	110
12.3. Functions: Building Reusable Code Blocks	110
12.4. Classes: Organizing Data and Behavior Together	115
12.5. Combining Functions and Classes	117
12.6. Function and Class Design Guidelines	117
12.7. Key Takeaways	118
12.8. Exercises	118
13. Working with Files	120
13.1. Introduction	120
13.2. Learning Objectives	120
13.3. Creating a Sample File	120
13.4. Reading and Writing Files	121
13.5. Exception Handling	122
13.6. Combining File Handling and Exception Handling	124
13.7. Working with Different File Formats	125
13.8. Key Takeaways	127

13.9. Exercises	128
14. Data Analysis with NumPy and Pandas	130
14.1. Introduction	130
14.2. Learning Objectives	130
14.3. Introduction to NumPy	130
14.4. Introduction to Pandas	141
14.5. Combining NumPy and Pandas	147
14.6. Key Takeaways	148
14.7. Further Reading	149
14.8. Exercises	149
III: Geospatial Programming with Python	151
15. Introduction to Geospatial Python	153
15.1. Introduction	153
15.2. The Geospatial Python Ecosystem	153
15.3. Understanding Library Relationships	154
15.4. Setting Up Your Environment	154
15.5. Verification and First Steps	156
15.6. Learning Path and Chapter Overview	156
15.7. Key Concepts to Remember	157
15.8. Getting Help and Resources	157
15.9. Next Steps	158
15.10. Exercises	158
16. Vector Data Analysis with GeoPandas	159
16.1. Introduction	159
16.2. Learning Objectives	159
16.3. Core Concepts	159
16.4. Installing GeoPandas	160
16.5. Creating GeoDataFrames	160
16.6. Reading and Writing Geospatial Data	161
16.7. Projections and Coordinate Reference Systems (CRS)	162
16.8. Spatial Measurements and Analysis	164
16.9. Visualizing Geospatial Data	166
16.10. Advanced Geometric Operations	170
16.11. Spatial Relationships and Queries	174
16.12. Best Practices and Performance Considerations	174
16.13. Key Takeaways	175
16.14. Exercises	175
17. Working with Raster Data Using Rasterio	177
17.1. Introduction	177
17.2. Learning Objectives	177
17.3. Installing Rasterio	178
17.4. Reading Raster Data	178
17.5. Visualizing Raster Data	181
17.6. Accessing and Manipulating Raster Bands	188
17.7. Writing Raster Data	189
17.8. Clipping Raster Data	190
17.9. Key Takeaways	193
17.10. Exercises	193

18. Multi-dimensional Data Analysis with Xarray	196
18.1. Introduction	196
18.2. Learning Objectives	196
18.3. Understanding Xarray's Data Model	197
18.4. Setting Up Your Environment	197
18.5. Loading and Exploring Real Climate Data	198
18.6. Working with DataArrays	199
18.7. Intuitive Data Selection and Indexing	201
18.8. Performing Operations on Multi-Dimensional Data	202
18.9. Data Visualization with Xarray	204
18.10. Working with Datasets: Multiple Variables	206
18.11. The Power of Label-Based Operations	207
18.12. Advanced Indexing Techniques	209
18.13. High-Level Computational Operations	210
18.14. Data Input and Output	213
18.15. Key Takeaways	214
18.16. Further Reading	215
18.17. Exercises	215
19. Raster Analysis with Rioxarray	217
19.1. Introduction	217
19.2. Learning Objectives	217
19.3. Setting Up Your Rioxarray Environment	217
19.4. Loading and Exploring Georeferenced Raster Data	218
19.5. Fundamental Geospatial Operations	221
19.6. Working with Spatial Dimensions and Resolution	222
19.7. Visualizing Geospatial Raster Data	224
19.8. Data Storage and File Management	228
19.9. Coordinate System Comparisons	230
19.10. Introduction to Band Math	232
19.11. Key Takeaways	235
19.12. Exercises	236
20. Interactive Visualization with Leafmap	238
20.1. Introduction	238
20.2. Learning Objectives	239
20.3. Installing and Setting Up Leafmap	239
20.4. Creating Interactive Maps	240
20.5. Changing Basemaps	243
20.6. Visualizing Vector Data	248
20.7. Creating Choropleth Maps	253
20.8. Visualizing GeoParquet Data	254
20.9. Visualizing PMTiles	256
20.10. Visualizing Raster Data	260
20.11. Accessing and Visualizing Maxar Open Data	267
20.12. Key Takeaways	274
20.13. Exercises	274
21. Geoprocessing with WhiteboxTools	278
21.1. Introduction	278
21.2. Learning Objectives	278

21.3.	Why Whitebox?	278
21.4.	Useful Resources for Whitebox	280
21.5.	Installing Whitebox	280
21.6.	Watershed Analysis	281
21.7.	LiDAR Data Analysis	294
21.8.	Key Takeaways	301
21.9.	Exercises	302
22.	<i>3D Mapping with MapLibre</i>	305
22.1.	Introduction	305
22.2.	Learning Objectives	305
22.3.	Useful Resources	305
22.4.	Installation and Setup	305
22.5.	Creating Interactive Maps	306
22.6.	Adding Map Controls	307
22.7.	Adding Layers	310
22.8.	Using MapTiler	312
22.9.	3D Mapping	313
22.10.	Visualizing Vector Data	320
22.11.	Visualizing Raster Data	330
22.12.	Adding Custom Components	332
22.13.	Visualizing PMTiles	339
22.14.	Adding DeckGL Layers	344
22.15.	Exporting to HTML	347
22.16.	Key Takeaways	347
22.17.	Exercises	347
23.	<i>Cloud Computing with Earth Engine and Geemap</i>	350
23.1.	Introduction	350
23.2.	Learning Objectives	350
23.3.	Introduction to Google Earth Engine	350
23.4.	Introduction to Interactive Maps and Tools	353
23.5.	The Earth Engine Data Catalog	358
23.6.	Earth Engine Data Types	360
23.7.	Earth Engine Raster Data	360
23.8.	Earth Engine Vector Data	362
23.9.	More Tools for Visualizing Earth Engine Data	365
23.10.	Vector Data Processing	373
23.11.	Raster Data Processing	376
23.12.	Exporting Earth Engine Data	382
23.13.	Creating Timelapse Animations	385
23.14.	Charting Earth Engine Data	390
23.15.	Key Takeaways	420
23.16.	Exercises	421
24.	<i>Hyperspectral Data Visualization with HyperCoast</i>	423
24.1.	Introduction	423
24.2.	Learning Objectives	423
24.3.	Environment Setup	424
24.4.	Finding Hyperspectral Data	424
24.5.	Downloading Hyperspectral Data	426

24.6.	Reading Hyperspectral Data	427
24.7.	Visualizing Hyperspectral Data	427
24.8.	Creating Image Cubes	429
24.9.	Interactive Slicing	430
24.10.	Interactive Thresholding	432
24.11.	Key Takeaways	433
24.12.	Exercises	433
25.	<i>High-Performance Geospatial Analytics with DuckDB</i>	435
25.1.	Introduction	435
25.2.	Learning Objectives	435
25.3.	Installation and Setup	436
25.4.	SQL Basics for Spatial Analysis	438
25.5.	Python API Integration	443
25.6.	Data Import	445
25.7.	Data Export	449
25.8.	Working with Geometries	451
25.9.	Spatial Relationships	454
25.10.	Spatial Joins	456
25.11.	Large-Scale Data Analysis	459
25.12.	Key Takeaways	466
25.13.	Exercises	467
26.	<i>Geospatial Data Processing with GDAL and OGR</i>	471
26.1.	Introduction	471
26.2.	Learning Objectives	471
26.3.	Installation and Setup	472
26.4.	Sample Datasets	472
26.5.	Understanding Your Data	472
26.6.	Coordinate Transformation	473
26.7.	Format Conversion	474
26.8.	Clipping and Masking	475
26.9.	Raster Analysis and Calculations	476
26.10.	Converting Between Raster and Vector	477
26.11.	Geometry Processing	478
26.12.	Managing Fields and Layers	479
26.13.	Tiling and Data Management	479
26.14.	Advanced Raster Processing	481
26.15.	Terrain Analysis	482
26.16.	Key Takeaways	488
26.17.	References and Further Reading	489
26.18.	Exercises	489
27.	<i>Building Interactive Dashboards with Voilà and Solara</i>	492
27.1.	Introduction	492
27.2.	Learning Objectives	493
27.3.	Installing Voilà and Solara	493
27.4.	Introduction to Hugging Face Spaces	493
27.5.	Creating a Basic Voilà Application	494
27.6.	Creating an Advanced Web Application with Solara	500
27.7.	Key Takeaways	505

27.8. Exercises	506
28. <i>Distributed Computing with Apache Sedona</i>	507
28.1. Introduction	507
28.2. Learning Objectives	508
28.3. Installing and Setting Up Apache Sedona	508
28.4. Downloading Sample Data	510
28.5. Core Concepts and Data Structures	510
28.6. Spatial Operations and Functions	513
28.7. Spatial Joins and Indexing	516
28.8. Advanced Spatial Analysis	519
28.9. Reading Vector Data	521
28.10. Visualizing Vector Data	524
28.11. Writing Vector Data	527
28.12. Reading Raster Data	527
28.13. Visualizing Raster Data	529
28.14. Raster Map Algebra	529
28.15. Raster Zonal Statistics	531
28.16. Writing Raster Data	532
28.17. Integration with GeoPandas	533
28.18. Real-World Use Cases	536
28.19. Key Takeaways	537
28.20. References and Further Reading	538
28.21. Exercises	539

Preface

Introduction

Geographic Information Systems (GIS) and geospatial analysis have become fundamental tools across numerous disciplines, from environmental science and urban planning to business analytics and public health. As the volume and complexity of geospatial data continue to grow exponentially, the ability to programmatically process, analyze, and visualize this data has become an essential skill for researchers, analysts, and professionals working with spatial information.

Python has emerged as the leading programming language for geospatial analysis, offering a rich ecosystem of libraries and tools that make complex spatial operations accessible to both beginners and experts. However, the path from Python novice to confident geospatial programmer can seem daunting, with numerous libraries to learn and concepts to master.

This book bridges that gap by providing a structured, practical approach to learning geospatial programming with Python. Rather than overwhelming you with advanced techniques from the start, we focus on building a solid foundation of essential skills that will serve you throughout your geospatial programming journey. Each chapter builds upon the previous ones, ensuring you develop both theoretical understanding and practical expertise.

The approach taken in this book is hands-on and example-driven. You'll work with real geospatial datasets, solve practical problems, and build projects that demonstrate the power of Python for geospatial analysis and visualization. By the end of this book, you'll have the confidence and skills to tackle your own geospatial programming challenges.

Who This Book Is For

This book is designed for a diverse audience of learners who want to harness the power of Python for geospatial analysis and visualization:

Students and Researchers in geography, environmental science, urban planning, data science, and related fields who need to analyze spatial data as part of their studies or research. No prior programming experience is assumed, though basic familiarity with computers and data analysis concepts is helpful.

GIS Professionals who currently use desktop GIS software and want to expand their toolkit with programming skills. If you've worked with ArcGIS, QGIS, or similar tools and want to automate workflows or perform analyses that are challenging in traditional GIS software, this book will help you make that transition.

Data Scientists and Analysts who work with location-based data and want to add spatial analysis capabilities to their skillset. If you're comfortable with Python basics but new to geospatial concepts, this book will introduce you to the spatial thinking and tools you need.

Software Developers interested in building applications that work with geospatial data. Whether you're developing web mapping applications, mobile apps with location features, or data processing pipelines, this book provides the foundation you need.

Self-Learners and Career Changers who are interested in the growing field of geospatial data science. The book assumes no prior knowledge of either Python programming or GIS concepts, making it accessible to motivated beginners.

Professionals in Government and Industry who need to incorporate spatial analysis into their work, such as urban planners, environmental consultants, market researchers, logistics coordinators, or public health officials.

The key requirement is curiosity and willingness to learn. While programming experience is helpful, it's not necessary. We start with the fundamentals and build up systematically.

What This Book Covers

This book is organized into three progressive sections that take you from software setup through Python fundamentals to advanced geospatial programming:

Software Setup prepares your development environment with everything you need for geospatial programming. You'll learn to install and configure essential tools including Miniconda for package management, VS Code for development, Git for version control, and cloud-based alternatives like Google Colab and JupyterLab. This section ensures you have a solid foundation before diving into programming.

Python Programming Fundamentals builds your core programming skills through seven comprehensive chapters. Starting with Python basics, you'll master variables and data types, data structures (lists, dictionaries, sets), string operations, control flow with loops and conditionals, functions and classes, file handling, and data analysis with NumPy and Pandas. These skills form the foundation for all geospatial programming tasks.

Geospatial Programming with Python comprises fourteen specialized chapters that transform you into a confident geospatial programmer:

- **Introduction to Geospatial Python** - Core concepts and the Python geospatial ecosystem
- **Vector Data Analysis with GeoPandas** - Working with points, lines, and polygons
- **Raster Data with Rasterio** - Processing satellite imagery and gridded datasets
- **Multi-dimensional Data Analysis with Xarray** - Handling complex scientific datasets
- **Raster Analysis with Rioxarray** - Advanced raster processing and analysis
- **Interactive Visualization with Leafmap** - Creating dynamic, interactive maps
- **Geoprocessing with WhiteboxTools** - Advanced spatial analysis operations
- **3D Mapping with MapLibre** - Building three-dimensional visualizations
- **Cloud Computing with Earth Engine and Geemap** - Leveraging Google Earth Engine for large-scale analysis
- **Hyperspectral Data Visualization with HyperCoast** - Working with hyperspectral data
- **High-Performance Geospatial Analytics with DuckDB** - High-performance spatial data processing
- **Geospatial Data Processing with GDAL and OGR** - Working with various geospatial data formats
- **Building Interactive Dashboards with Solara** - Creating interactive dashboards for geospatial applications
- **Distributed Computing with Apache Sedona** - Processing large geospatial datasets in a distributed environment

Each chapter follows a consistent structure:

- Clear concept explanations with real-world context
- Step-by-step code examples with detailed annotations
- Hands-on exercises using authentic geospatial datasets
- Common pitfalls and troubleshooting guidance
- References to additional resources and further reading

The progression is carefully designed so that each chapter builds upon previous concepts while introducing new capabilities, ensuring you develop both breadth and depth in geospatial programming.

Getting the Most Out of This Book

To maximize your learning experience with this book, consider the following recommendations:

Set Up a Proper Development Environment: Install Python and the required libraries as described in the first section of the book. A well-configured environment will save you time and frustration throughout your learning journey. Consider using conda or uv to manage your Python packages, as this simplifies the installation of geospatial libraries.

Follow Along with Code Examples: This book is designed to be interactive. Don't just read the code—type it out, run it, and experiment with modifications. Understanding comes through practice, and each example builds skills you'll need later.

Work Through the Exercises: Each chapter includes exercises designed to reinforce the concepts you've learned. These are not optional extras—they are an integral part of the learning process. Start with the guided exercises, then challenge yourself with your own projects.

Use Real Data: While the book provides datasets for examples and exercises, try applying the techniques to data from your own field or interests. This will help you understand how the concepts apply to real-world scenarios and build confidence in your abilities.

Build Projects: As you progress through the book, consider working on a personal project that interests you. This could be analyzing data from your research, creating maps for your community, or solving a problem you've encountered in your work.

Be Patient with Yourself: Programming can be frustrating, especially when you're learning. Expect to encounter errors, spend time debugging, and occasionally feel stuck. This is normal and part of the learning process. Take breaks when needed, and remember that expertise develops gradually through consistent practice. If you get stuck, don't hesitate to ask for help on the book's GitHub repository.

Keep Practicing: The skills in this book require regular practice to maintain and develop. Set aside time regularly to work on geospatial programming projects, even if they're small ones.

Conventions Used in This Book

This book uses several conventions to help you navigate the content and understand the code examples:

Code Formatting: All Python code appears in monospaced font within code blocks. When code appears within regular text, it is formatted `like this`. File and directory names are also formatted in monospaced font.

Code Examples: Most code examples are complete and runnable. They include comments explaining the key concepts and techniques being demonstrated. Line numbers may be included for reference in the accompanying text.

```
# This is an example of a code block
import leafmap
m = leafmap.Map()
m.add_basemap("OpenTopoMap") # add a basemap to the map
m
```

Command Line Instructions: Commands to be entered at the command line or terminal are shown with a `$` prompt:

```
$ pip install leafmap
$ python script.py
```

Downloading the Code Examples

All code examples, datasets, and supplementary materials for this book are freely available on GitHub:

<https://github.com/giswqs/intro-gispro>

To download the materials, you can use one of the following methods:

- **Clone the repository** (if you have Git installed):

```
$ git clone https://github.com/giswqs/intro-gispro.git
```

- **Download as ZIP** (if you prefer not to use Git):
 - Visit the GitHub repository page
 - Click the green **Code** button
 - Select **Download ZIP**
 - Extract the files to your preferred location
- **Browse individual files** online through the GitHub interface if you only need specific examples

The repository is regularly updated with corrections, improvements, and additional examples. Check back periodically for updates, or **watch** the repository on GitHub to be notified of changes.

If you find errors in the code or have suggestions for improvements, please open an issue or submit a pull request on GitHub. Community contributions help make this resource better for everyone.

Video Tutorials

Complementing the written content, this book is supported by a comprehensive series of video tutorials that walk through key concepts and provide additional examples:

<https://tinyurl.com/intro-gispro-videos>

The videos are designed to complement, not replace, the written material. They're particularly helpful for:

- Visual learners who benefit from seeing code being written and executed
- Understanding complex concepts through multiple explanations
- Learning about the development workflow and best practices
- Seeing how to approach problems and debug issues

The playlist is organized to follow the book's structure. You can watch them in order as you progress through the book, or jump to specific topics as needed.

The videos were created in Fall 2024 when I was teaching the [Introduction to GIS Programming](https://geog-312.gishub.org)¹ course at the University of Tennessee. Although the course has concluded, the videos remain relevant and can be used as a reference for the book. Additional videos will be added in the future.

¹<https://geog-312.gishub.org>

Get in Touch

I welcome feedback, questions, and suggestions from readers. Your input helps improve the book and makes it more useful for the geospatial programming community.

For book-related questions and discussions:

- GitHub Issues: <https://github.com/giswqs/intro-gispro/issues>
- GitHub Discussions: <https://github.com/giswqs/intro-gispro/discussions>

Types of feedback that are particularly helpful:

- Errors or unclear explanations in the text or code
- Suggestions for additional examples or use cases
- Ideas for new topics or chapters
- Reports of compatibility issues with different operating systems or library versions
- Success stories of how you've applied the techniques from the book

Acknowledgments

This book would not have been possible without the contributions and support of many individuals and the broader open-source geospatial community.

The Open-Source Community: This book builds upon the incredible work of countless open-source developers who have created and maintained the Python geospatial ecosystem. Special thanks to the developers and maintainers of NumPy, Pandas, GeoPandas, Rasterio, Xarray, Rioxarray, Folium, ipyleaflet, MapLibre, GDAL, and the many other libraries that make geospatial programming accessible.

Students and Colleagues: The questions, challenges, and insights from students in my geospatial programming courses at the University of Tennessee have shaped the approach and content of this book. Their feedback on what works and what doesn't has been invaluable in creating materials that truly serve learners.

Research Collaborators: Colleagues and collaborators in the geospatial research community have provided real-world use cases, datasets, and problem scenarios that inform the practical examples throughout the book.

Family and Friends: Writing a technical book requires significant time and focus. I'm grateful for the patience and support of family and friends who understood the many evenings and weekends dedicated to this project.

The Broader GIS Community: The geospatial field is built on a foundation of sharing knowledge and tools. This book is part of that tradition, and I'm honored to contribute to the resources available for learning geospatial programming.

This book was written using [MyST Markdown](https://mystmd.org)² and compiled using [Typst](https://github.com/typst/typst)³ with the [min-book](https://github.com/mayconfmelo/min-book)⁴ template. Credits to developers and maintainers of the Typst and MyST Markdown projects. Special thanks to [Maycon F. Melo](https://github.com/mayconfmelo)⁵ for the min-book template and their help with customizing the template for this book.

²<https://mystmd.org>

³<https://github.com/typst/typst>

⁴<https://github.com/mayconfmelo/min-book>

⁵<https://github.com/mayconfmelo>

Any errors or omissions in this book remain my responsibility. I'm committed to addressing issues and improving the content based on reader feedback.

About the Author

Dr. Qiusheng Wu is an Associate Professor and the Director of Graduate Studies in the Department of Geography & Sustainability at the University of Tennessee, Knoxville. He also serves as an Amazon Scholar. Dr. Wu's research focuses on geospatial data science and open-source software development, with an emphasis on leveraging big geospatial data and cloud computing to study environmental change, particularly surface water and wetland inundation dynamics. He is the creator of several widely used open-source Python packages, including [geemap](https://geemap.org)⁶, [leafmap](https://leafmap.org)⁷, [segment-geospatial](https://samgeo.gishub.org)⁸, and [geoai](https://opengeoai.org)⁹, which support advanced geospatial analysis and interactive visualization. His open-source work is available at the [Open Geospatial Solutions](https://github.com/opengeos)¹⁰ on GitHub.

Licensing and Copyright

This book embraces the principles of open science and open education. To support transparency, learning, and reuse, the **code examples** in this book are released under a [Creative Commons Attribution 4.0 International \(CC BY 4.0\)](https://creativecommons.org/licenses/by/4.0/) license. This means you are free to copy, modify, and distribute the code, even for commercial purposes, as long as appropriate credit is given.

Please attribute code usage by citing the book or linking to the GitHub repository:

Wu, Q. (2025). *Introduction to GIS Programming: A Practical Python Guide to Open Source Geospatial Tools*. <https://gispro.gishub.org>

While the code is freely available, the **text, figures, and images** in this book are **copyrighted** by the author and may not be reproduced, redistributed, or modified without explicit permission. This includes all written content, custom diagrams, and embedded visualizations unless otherwise noted.

If you wish to reuse or adapt any non-code material from the book—for example, for teaching, presentations, or publications—please contact the author to request permission.

This dual licensing approach helps balance open access to learning materials with the protection of original creative work. Thank you for respecting these terms and supporting the open-source geospatial community.

⁶<https://geemap.org>

⁷<https://leafmap.org>

⁸<https://samgeo.gishub.org>

⁹<https://opengeoai.org>

¹⁰<https://github.com/opengeos>

Section I: Software Setup

Chapter 1. Overview of Software Tools

1.1. Introduction

Successful geospatial programming requires more than just knowing Python—it demands a carefully selected toolkit of software that works together seamlessly. Just as a carpenter needs the right combination of tools to build a house, geospatial programmers need an integrated set of applications that handle everything from managing Python packages to creating interactive maps.

This chapter introduces six essential tools that form the foundation of modern geospatial programming workflows. These tools address the core challenges you'll face: managing complex software dependencies, writing and debugging code efficiently, tracking changes and collaborating with others, and running analyses in both local and cloud environments.

1.2. Learning Objectives

By the end of this chapter, you should be able to:

- Understand the essential software tools needed for professional geospatial programming workflows
- Explain how different tools complement each other in a unified geospatial development environment
- Choose appropriate tools for specific geospatial programming tasks and project requirements
- Recognize the benefits of an integrated toolkit approach

1.3. Essential Software Tools

Modern geospatial programming relies on six fundamental categories of tools, each addressing specific workflow needs:

1.3.1. Package Management: Conda

What it is: [Conda](https://docs.conda.io)¹¹ is a cross-platform package manager that excels at handling complex dependencies, especially for scientific computing and geospatial libraries.

Why it's essential: Geospatial programming involves complex libraries like GDAL, GEOS, and PROJ that have intricate system-level dependencies. Conda handles these dependencies automatically, creates isolated environments for different projects, and ensures reproducible setups across different computers.

Key benefits:

- Manages both Python packages and underlying system libraries
- Creates isolated environments to prevent version conflicts
- Handles complex geospatial dependencies automatically
- Enables reproducible development environments

1.3.2. Code Development: Visual Studio Code

What it is: [Visual Studio Code](https://code.visualstudio.com)¹² is a lightweight yet powerful code editor that provides exceptional support for Python development, Jupyter notebooks, and geospatial programming workflows.

¹¹<https://docs.conda.io>

¹²<https://code.visualstudio.com>

Why it's essential: VS Code combines the simplicity of a text editor with the power of an IDE, offering intelligent code completion, debugging, Git integration, and native Jupyter notebook support—all essential for efficient geospatial programming.

Key benefits:

- Intelligent Python code completion and debugging
- Native Jupyter notebook support for interactive analysis
- Built-in Git integration for version control
- Extensive extension ecosystem for geospatial tools
- Remote development capabilities for cloud computing

1.3.3. Version Control: Git

What it is: [Git](https://git-scm.com)¹³ is a distributed version control system that tracks changes in files and enables collaborative development among multiple contributors.

Why it's essential: Geospatial projects often involve complex analyses, multiple data sources, and collaborative research. Git provides a safety net for your work, enables experimentation without fear of losing progress, and facilitates seamless collaboration with colleagues.

Key benefits:

- Complete history of all project changes
- Safe experimentation through branching and merging
- Seamless collaboration and code sharing
- Professional development workflow standards
- Integration with GitHub for remote backup and sharing

1.3.4. Cloud Computing: Google Colab

What it is: [Google Colab](https://colab.research.google.com)¹⁴ is a cloud-based Jupyter notebook environment that provides free access to computational resources, including GPUs and TPUs.

Why it's essential: Cloud computing democratizes access to powerful computing resources and eliminates setup complexity. Colab provides instant access to a fully configured geospatial programming environment with no local installation required.

Key benefits:

- Zero setup—start coding immediately in your browser
- Free access to GPU/TPU for machine learning applications
- Pre-installed geospatial libraries and tools
- Easy sharing and collaboration through Google Drive integration
- No hardware limitations for processing large datasets

1.3.5. Interactive Analysis: JupyterLab

What it is: [JupyterLab](https://jupyterlab.readthedocs.io)¹⁵ is a web-based interactive development environment that extends traditional Jupyter notebooks with a flexible, multi-document interface.

¹³<https://git-scm.com>

¹⁴<https://colab.research.google.com>

¹⁵<https://jupyterlab.readthedocs.io>

Why it's essential: Geospatial analysis is inherently exploratory and iterative. JupyterLab provides the perfect environment for combining code, visualizations, and documentation in an interactive workflow that supports the experimental nature of spatial data analysis.

Key benefits:

- Combines code, visualizations, and documentation seamlessly
- Flexible workspace for managing multiple files and notebooks
- Excellent support for interactive data exploration
- Integration with various file formats and data sources
- Extensible architecture for specialized geospatial tools

1.3.6. Containerization: Docker

What it is: [Docker](https://www.docker.com)¹⁶ provides a way to package entire development environments—including the operating system, Python, libraries, and code—into portable containers.

Why it's essential: Geospatial programming often involves complex software dependencies that can be challenging to install and configure. Docker ensures your code runs consistently across different computers and provides an easy way to share reproducible environments.

Key benefits:

- Consistent environments across different computers
- Easy setup and distribution of complex geospatial software stacks
- Isolation prevents conflicts between different projects
- Reproducible research through containerized environments
- Seamless deployment from development to production

1.4. Tool Integration and Workflow

These tools work together to create a comprehensive geospatial programming environment:

Development Workflow:

1. **Conda** manages your Python environment and installs geospatial libraries
2. **VS Code** provides your primary development interface with intelligent code editing
3. **Git** tracks all changes and enables collaboration
4. **JupyterLab** supports interactive analysis and data exploration
5. **Google Colab** provides cloud-based computing for resource-intensive tasks
6. **Docker** ensures reproducible environments and easy deployment

Collaboration Flow:

- Write code in **VS Code** with intelligent completion and debugging
- Experiment and visualize data in **JupyterLab** notebooks
- Track changes and collaborate using **Git** and GitHub
- Share cloud-based analyses through **Google Colab**
- Distribute reproducible environments using **Docker** containers
- Manage all dependencies consistently with **Conda**

¹⁶<https://www.docker.com>

1.5. Running Code Examples

You can run the code examples from this book using several approaches:

Cloud Options (No Installation Required):

- **Binder:** <https://mybinder.org/v2/gh/giswqs/intro-gispro/HEAD>
- **Google Colab:** <https://colab.research.google.com/github/giswqs/intro-gispro/blob/main>

Local Development:

- **Docker:** `docker run -it -p 8888:8888 -v $(pwd):/app/workspace giswqs/pygis:book`
- **Conda Environment:** Follow the installation instructions in subsequent chapters

You will learn how to use these tools in the following chapters.

1.6. Key Takeaways

Integrated Approach: The power of these tools comes from using them together, not in isolation. Each tool addresses specific challenges while complementing the others to create a complete workflow.

Progressive Learning: You don't need to master all tools immediately. Start with the basics (Conda and VS Code), then gradually incorporate additional tools as your projects become more complex.

Reproducibility: This toolkit emphasizes reproducible research practices through environment management, version control, and containerization—essential for professional geospatial programming.

Flexibility: The combination of local development tools (VS Code, JupyterLab) and cloud platforms (Google Colab, Docker) provides flexibility to work in any environment.

Professional Standards: These tools represent industry-standard practices used by professional developers and researchers worldwide, ensuring your skills are transferable and valuable.

Mastering this integrated toolkit will significantly enhance your productivity, enable effective collaboration, and provide the foundation for tackling complex geospatial programming challenges throughout your career. Happy coding!

Chapter 2. Introduction to Python Package Management

2.1. Introduction

One of the biggest challenges in Python programming—especially in data science and geospatial analysis—is managing the complex web of software dependencies. Different projects require different versions of Python packages, and sometimes these requirements conflict with each other. Installing geospatial libraries (e.g., GDAL, GEOS, PROJ) can be particularly challenging because they often depend on complex underlying C/C++ libraries for tasks like coordinate transformations and data format support.

This is where package managers like **Conda** and **uv** become essential tools. Think of them as smart assistants that handle the tedious work of finding, downloading, and installing the right versions of software packages while ensuring everything works together harmoniously.

Why package management matters for geospatial programming:

- **Complex dependencies:** Geospatial libraries like GDAL and Rasterio have intricate dependencies on system libraries
- **Version compatibility:** Different projects may require different versions of the same library
- **Environment isolation:** Keep project dependencies separate to avoid conflicts
- **Reproducibility:** Ensure your code works the same way on different computers
- **Easy collaboration:** Share exact environment specifications with colleagues

Conda is a cross-platform package manager that excels at handling complex dependencies, especially for scientific computing. It can install not just Python packages, but also system libraries, compilers, and other tools that geospatial packages often require.

uv is a newer, extremely fast package manager written in Rust that focuses on Python packages. While it primarily works with [PyPI](https://pypi.org)¹⁷ (the Python Package Index), it offers significant speed improvements over traditional tools like pip.

This chapter will teach you to use both tools effectively, helping you create reliable, reproducible development environments for your geospatial programming projects.

2.2. Learning Objectives

By the end of this chapter, you should be able to:

- Understand the importance of package management and virtual environments in Python development
- Install and configure Conda (Miniconda) for geospatial programming
- Create, manage, and activate virtual environments for different projects
- Install and manage Python packages using conda and mamba
- Use uv as a fast alternative for package management
- Apply best practices for reproducible environment management
- Troubleshoot common package management issues
- Share and recreate environments for collaboration

¹⁷<https://pypi.org>

2.3. Installing Conda (Miniconda)

2.3.1. Why Miniconda?

Miniconda¹⁸ is the recommended way to install Conda. Unlike the full **Anaconda**¹⁹ distribution, which comes with hundreds of pre-installed packages, Miniconda includes only the essential components: conda, Python, and a few basic packages. This minimal approach gives you more control over what gets installed in your environment.

2.3.2. Installation

To install Miniconda, visit the Miniconda installation page: <https://tinyurl.com/miniconda-install>. Depending on your operating system, you can choose the appropriate graphical installer or the command line installer. The instructions below are for the command line installer.

2.3.2.1. Windows

Open Windows PowerShell as administrator and run the following command:

```
curl https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.exe -  
o .\miniconda.exe  
start /wait "" .\miniconda.exe /S  
del .\miniconda.exe
```

After the installation is complete, you can close the PowerShell window. Go to the Start menu and search for “Anaconda Prompt” to open a new terminal.

2.3.2.2. macOS

If you are using macOS with Apple Silicon, run the following command:

```
mkdir -p ~/miniconda3  
curl https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-arm64.sh -o ~/miniconda3/miniconda.sh  
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3  
rm ~/miniconda3/miniconda.sh
```

If you are using macOS with Intel, run the following command:

```
mkdir -p ~/miniconda3  
curl https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.sh -o ~/miniconda3/miniconda.sh  
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3  
rm ~/miniconda3/miniconda.sh
```

¹⁸<https://www.anaconda.com/docs/getting-started/miniconda/main>

¹⁹<https://www.anaconda.com>

After the installation is complete, close the terminal and open a new terminal. Run the following commands to initialize conda:

```
source ~/miniconda3/bin/activate
conda init --all
```

2.3.2.3. Linux

If you are using Linux, run the following command:

```
mkdir -p ~/miniconda3
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O ~/miniconda3/miniconda.sh
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
rm ~/miniconda3/miniconda.sh
```

After the installation is complete, close the terminal and open a new terminal. Run the following commands to initialize conda:

```
source ~/miniconda3/bin/activate
conda init --all
```

2.3.3. Verifying Installation

After installation, open a new terminal or command prompt and verify conda is working:

```
conda --version
conda info
```

You should see version information and system details. If you get a “command not found” error, conda may not be in your PATH (see troubleshooting section below).

By default, conda will automatically activate the base environment when you open a new terminal. It is recommended to disable automatic activation of the base environment to avoid accidentally installing packages into the base environment. You can do this by running the following command:

```
conda config --set auto_activate_base false
```

2.4. Understanding Conda Concepts

Before diving into commands, it’s important to understand key concepts that make conda powerful for geospatial programming.

2.4.1. Environments

Think of a conda environment as an isolated workspace for a specific project. Each environment has its own Python installation and packages, preventing conflicts between different projects.

Why use environments:

- **Isolation:** Keep project dependencies separate
- **Reproducibility:** Create identical setups on different machines
- **Experimentation:** Test new packages without affecting other projects
- **Collaboration:** Share exact environment specifications with teammates

2.4.2. Channels

Channels are repositories where conda packages are stored. Different channels may have different versions or specialized packages.

There are two main channels for conda packages:

- **defaults:** Anaconda's main channel. It is the default channel for conda. However, the packages in this channel are often outdated.
- **conda-forge:** A community-driven channel with many geospatial packages. The packages in this channel are often up-to-date. It is the recommended channel for installing geospatial packages.

2.5. Creating Your First Geospatial Environment

Let's create a dedicated environment for geospatial programming with all the essential packages:

```
# Create a new environment named 'geo' with Python 3.12
conda create -n geo python=3.12

# Activate the environment
conda activate geo

# Install mamba for faster package management
conda install -n base mamba -c conda-forge

# Install essential packages for geospatial programming
mamba install -c conda-forge pygis
```

What each command does:

1. **conda create -n geo python=3.12** : Creates a new environment called “geo” with Python 3.12
2. **conda activate geo** : Switches to the geo environment
3. **conda install -n base mamba -c conda-forge** : Installs mamba (a faster conda) in the base environment
4. **mamba install -c conda-forge pygis** : Uses mamba to quickly install geospatial packages from conda-forge

After running these commands, you'll have a complete environment ready for geospatial programming!

2.6. Troubleshooting Conda

If you're on Windows and conda commands aren't recognized in your terminal, you may need to manually add conda to your system PATH. This is a common issue when the installer doesn't automatically configure the PATH variable.

Method 1: Using Environment Variables GUI

1. **Open the Start Menu** and search for “Environment Variables”
2. **Click on “Edit the system environment variables”** (see [Figure 1](#))
3. In the System Properties window, click “**Environment Variables**”
4. Under “System Variables,” find the **Path** variable and select it
5. Click “**Edit**” and then “**New**”
6. Add these paths (replace <YourUsername> with your actual username):
 - C:\Users\<YourUsername>\miniconda3
 - C:\Users\<YourUsername>\miniconda3\Scripts
 - C:\Users\<YourUsername>\miniconda3\Library\bin
7. Click “**OK**” to close all windows
8. **Restart your terminal** or command prompt

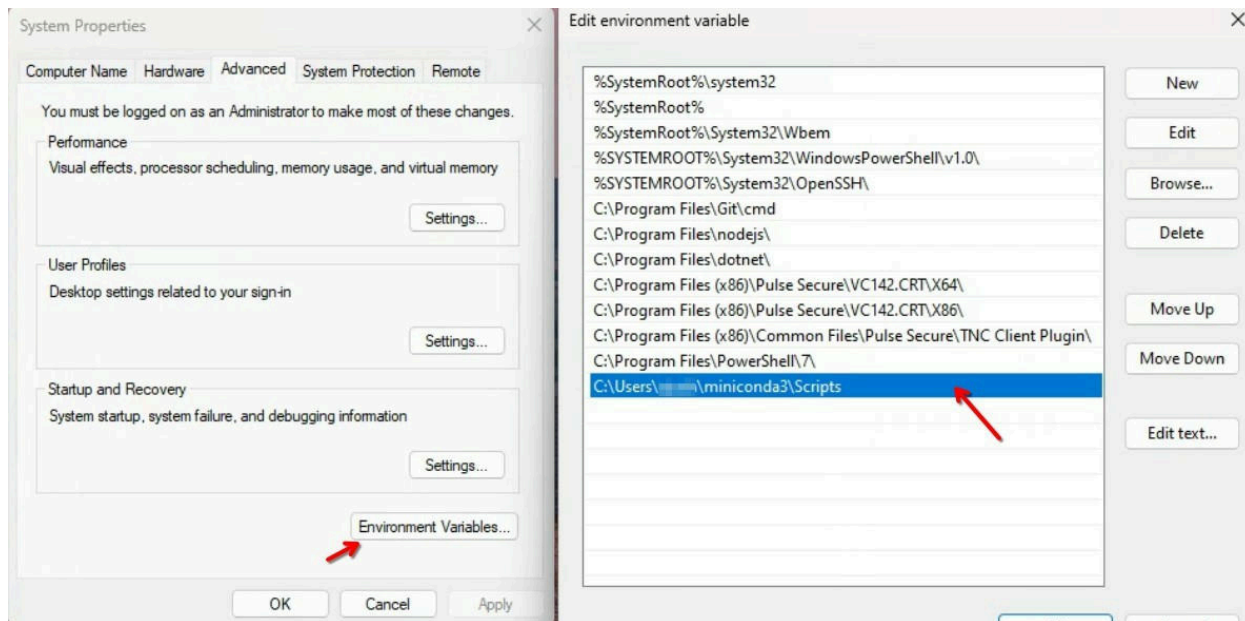


Figure 1: Add conda to the PATH environment variable.

Method 2: Using Anaconda Prompt

If the PATH method doesn't work, you can always use the **Anaconda Prompt** which comes with the Miniconda installation:

1. Open the Start Menu
2. Search for “Anaconda Prompt”
3. This terminal has conda pre-configured and ready to use

Method 3: Initialize conda in your shell

Open a command prompt as administrator and run:

```
conda init cmd.exe
```

Then restart your command prompt.

2.7. Essential Conda Commands

Now that you understand the concepts, let's explore the essential commands you'll use regularly for geospatial programming. These commands form the foundation of effective package and environment management.

2.7.1. Creating and Managing Environments

Environments are the foundation of reproducible Python development. Here are the key commands with practical examples:

Create a new environment:

```
# Basic environment with specific Python version
conda create -n myenv python=3.12

# Environment with multiple packages from the start
conda create -n geoenv python=3.12 numpy pandas matplotlib

# Create environment with packages from specific channels
conda create -n geoenv2 python=3.12 -c conda-forge geopandas
```

Activate an environment:

```
conda activate myenv
```

After activation, your terminal prompt will show the environment name, indicating you're working within that environment.

Deactivate the current environment:

```
conda deactivate
```

This returns you to the base environment.

List all environments:

```
conda env list
# or
conda info --envs
```

The active environment will be marked with an asterisk ().*

Remove an environment:

```
# Remove entire environment and all its packages
conda remove -n myenv --all

# Alternative method using env remove
conda env remove -n myenv
```

Clone an existing environment:

```
# Create a copy of an existing environment
conda create -n newenv --clone oldenv
```

2.7.2. Installing and Managing Packages

Package management is where conda really shines, especially for geospatial libraries with complex dependencies:

Install packages in the current environment:

```
# Install a package from the main channel
conda install numpy

# Install multiple packages from the main channel
conda install scipy matplotlib seaborn

# Install specific versions
conda install numpy=1.24.0 pandas>=1.5.0
```

Install packages in a specific environment:

```
# Install without activating the environment
conda install -n myenv pandas

# Useful for setting up environments remotely
conda install -n geovenv -c conda-forge geopandas rasterio
```

Install packages from specific channels:

```
# Install from conda-forge (recommended for geospatial packages)
conda install -c conda-forge geopandas
```

Update packages:

```
# Update all packages in current environment
conda update --all
```

```
# Update specific packages
conda update numpy pandas

# Update conda itself
conda update conda
```

Search and inspect packages:

```
# Search for packages
conda search scikit-learn
conda search "*gdal*" # wildcard search

# Get package information
conda search -c conda-forge geopandas --info

# List all installed packages
conda list

# List packages matching a pattern
conda list "*geo*"
```

Remove packages:

```
# Remove a single package
conda remove numpy

# Remove multiple packages
conda remove scipy matplotlib

# Remove packages and their dependencies (if not needed by others)
conda remove numpy --all
```

2.7.3. Using Mamba (Faster Package Management)

[Mamba](https://mamba.readthedocs.io)²⁰ is a drop-in replacement for conda that provides significant speed improvements, especially when installing complex packages like those used in geospatial programming.

Why use Mamba:

- **Much faster:** Uses C++ and parallel processing for dependency resolution
- **Same interface:** All conda commands work with mamba
- **Better error messages:** More informative when conflicts occur
- **Recommended for geospatial:** Handles complex dependency trees more efficiently

Install mamba:

²⁰<https://mamba.readthedocs.io>


```
# Install mamba in the base environment (do this once)
conda install -n base mamba -c conda-forge
```

Use mamba instead of conda:

```
# These commands are much faster with mamba
mamba create -n geofast python=3.12
mamba activate geofast
mamba install -c conda-forge geopandas rasterio geemap leafmap

# All conda commands work with mamba
mamba list
mamba update --all
mamba remove geopandas
```

2.7.4. Environment Files for Reproducibility

One of the most powerful features for collaboration is the ability to share exact environment specifications:

Export environment to file:

```
# Export all packages and versions
conda env export > environment.yml

# Export with specific name
conda env export -n myenv > myenv.yml
```

Create environment from file:

```
# Create environment from exported file
conda env create -f environment.yml

# Create with different name
conda env create -f environment.yml -n newname
```

Example environment.yml for geospatial programming:

```
name: geo
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.12
  - geopandas
  - rasterio
```

- `geemap`
- `leafmap`
- `jupyterlab`

2.8. Introducing uv: The Fast Alternative

`uv`²¹ is a revolutionary Python package manager written in Rust that offers dramatic speed improvements over traditional tools like `pip`. While `conda` excels at managing system-level dependencies and complex scientific packages, `uv` is perfect for pure Python packages available on PyPI.

When to use uv vs conda:

- **Use conda for:** Geospatial packages with C/C++ dependencies (GDAL, GEOS, PROJ)
- **Use uv for:** Pure Python packages, web development, and fast prototyping

2.8.1. Installing uv

On macOS and Linux:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

On Windows:

```
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"
```

Using pip (if you already have Python):

```
pip install uv
```

2.8.2. Basic uv Usage

Create and manage environments:

```
# Navigate to your project directory
cd /path/to/your/project

# Create a virtual environment
uv venv

# Create with specific Python version
uv venv --python 3.12

# Activate the environment (varies by OS)
# On macOS/Linux:
```

²¹<https://docs.astral.sh/uv>

```
source .venv/bin/activate
# On Windows:
.venv\Scripts\activate
```

Install packages with blazing speed:

```
# Install packages
uv pip install jupyterlab leafmap

# Install from requirements file
uv pip install -r requirements.txt

# Run Python directly in the environment
uv run python script.py

# Run Jupyter directly
uv run jupyter lab
```

2.8.3. uv vs pip Performance

uv can be **10-100x faster** than pip for package installation, making it ideal for:

- **Rapid prototyping:** Quickly set up environments for testing ideas
- **CI/CD pipelines:** Faster builds and deployments (e.g., GitHub Actions)
- **Large projects:** Installing many dependencies quickly
- **Development workflows:** Frequent package installations

2.9. Best Practices for Package Management

As you develop your geospatial programming skills, following these best practices will save you time and prevent frustration:

2.9.1. Environment Management

- **Use descriptive names:** Name environments with meaningful names but not too long.
- **Keep environments focused:** Don't install everything in one environment
- **Use environment files:** Always export environment specifications for reproducibility
- **Regular cleanup:** Remove unused environments to save disk space

2.9.2. Package Installation

- **Prefer conda-forge:** Most geospatial packages are best installed from conda-forge
- **Use mamba for speed:** Install mamba and use it for package operations
- **Specify channels explicitly:** Use `-c conda-forge` to avoid conflicts
- **Pin critical versions:** Specify exact versions for production environments

2.9.3. Collaboration and Reproducibility

- **Share environment files:** Include `environment.yml` in your project repositories
- **Document installation steps:** Include clear setup instructions in README files
- **Test on fresh environments:** Verify your code works in clean environments
- **Use version pinning carefully:** Balance stability with updates

2.9.4. Troubleshooting Tips

- **Update conda first:** Many issues are resolved by updating conda/mamba
- **Clear cache:** Use `conda clean --all` to clear package cache
- **Check channels:** Verify you're using the right channels for packages
- **Read error messages:** Conda provides detailed information about conflicts

2.10. Key Takeaways

Understanding package management is crucial for successful geospatial programming. Here are the essential concepts to remember:

Core Concepts:

- **Virtual environments** isolate project dependencies and prevent conflicts
- **Channels** are repositories where packages are stored (conda-forge is essential for geospatial work)
- **Package managers** (conda, mamba, uv) automate the complex task of dependency resolution

Tool Selection:

- **Conda/Mamba:** Best for geospatial packages with complex system dependencies
- **uv:** Excellent for pure Python packages and rapid development
- **Hybrid approach:** Use both tools together for maximum efficiency

Essential Workflows:

- Create dedicated environments for each project
- Use mamba for faster package installation
- Export environment specifications for reproducibility
- Regularly update packages and clean unused environments

Mastering these package management tools will make your geospatial programming journey smoother and more productive. You should avoid installing packages into the system Python environment or the conda base environment. Instead, create a new environment for each project. Otherwise, if the system Python environment or conda base environment is corrupted, you may not be able to use the system Python or conda.

2.11. Exercises

These hands-on exercises will help you practice essential package management skills for geospatial programming.

2.11.1. Exercise 1: Setting Up Your First Geospatial Environment

Objective: Create a complete geospatial programming environment from scratch.

Tasks:

1. Create a new conda environment called `gispro` with Python 3.12
2. Activate the environment and install mamba
3. Use mamba to install these essential packages from conda-forge:
 - `geopandas`
 - `rasterio`
 - `geemap`
 - `leafmap`
 - `jupyterlab`
4. Verify the installation by listing all installed packages
5. Test the environment by running Python and importing each package

2.11.2. Exercise 2: Environment Management and Reproducibility

Objective: Practice sharing and recreating environments.

Tasks:

1. Export your `gispro` environment to a file called `gispro-environment.yml`
2. Create a second environment called `gispro-copy` from this exported file
3. Compare the two environments using `conda list`
4. Add the `maplibre` package to `gispro-copy`
5. Export the updated environment with a new name
6. Remove the original `gispro` environment

2.11.3. Exercise 3: Exploring uv for Fast Development

Objective: Experience uv's speed advantages for Python-only packages.

Tasks:

1. Install uv on your system (if not already installed)
2. Create a new directory called `fast-geo-project`
3. Use uv to create a virtual environment with Python 3.12
4. Install these packages using uv:
 - `geopandas`
 - `rasterio`
 - `geemap`
 - `leafmap`
 - `jupyterlab`
5. Time the installation and compare to conda/pip
6. Run JupyterLab using `uv run jupyter lab`

Chapter 3. Setting Up Visual Studio Code

3.1. Introduction

In the world of geospatial programming, having the right development environment can make the difference between frustration and productivity. **Visual Studio Code**²² (VS Code) is the go-to code editor for Python developers, data scientists, and geospatial programmers worldwide—and for good reasons.

Think of VS Code as your digital workshop: a place where you'll write code, debug problems, manage files, explore data, create visualizations, and collaborate with others. Unlike simple text editors, VS Code understands your code, helps you write it more efficiently, and integrates seamlessly with the tools you'll use for geospatial analysis.

Why VS Code excels for geospatial programming:

- **Python mastery:** Exceptional support for Python with intelligent code completion, debugging, and testing
- **Jupyter integration:** Run Jupyter notebooks directly in the editor—perfect for geospatial data exploration
- **Git version control:** Built-in Git support for tracking changes and collaborating on projects
- **Extension ecosystem:** Thousands of extensions, including specialized tools for Python and Jupyter
- **Remote development:** Work on cloud servers or remote machines as if they were local
- **Multi-language support:** Handle Python, JavaScript, SQL, and markup languages in one place
- **Free and open-source:** Professional-grade tools available to everyone

What makes VS Code special for our work: Unlike heavyweight IDEs that can be overwhelming, or basic editors that lack functionality, VS Code strikes the perfect balance. It's lightweight enough to start quickly, yet powerful enough to handle complex geospatial projects involving multiple data formats, large datasets, and sophisticated analyses.

Whether you're processing satellite imagery, analyzing GPS tracks, building web maps, or developing geospatial web applications, VS Code provides the tools and flexibility you need to work efficiently and effectively. I have been using VS Code for geospatial programming for several years, and it has become an indispensable tool for my daily work.

3.2. Learning Objectives

By the end of this chapter, you should be able to:

- Install and configure Visual Studio Code for geospatial programming
- Set up essential extensions for Python and Jupyter development
- Navigate VS Code efficiently using keyboard shortcuts and features
- Configure VS Code for optimal geospatial development workflows
- Use integrated terminals and version control features
- Customize the editor to match your preferences and workflow
- Troubleshoot common setup issues and optimize performance
- Apply best practices for organizing and managing geospatial projects

²²<https://code.visualstudio.com>

3.3. Installing Visual Studio Code

VS Code is available for Windows, macOS, and Linux, and the installation process is straightforward on all platforms.

3.3.1. Download and Installation

1. **Visit the official website:** <https://code.visualstudio.com>
2. **Download the installer:**
 - **Windows:** Download the Windows installer (.exe)
 - **macOS:** Download the macOS installer (.dmg)
 - **Linux:** Choose from .deb (Ubuntu/Debian) or .rpm (Red Hat/CentOS) packages
3. **Run the installer:**
 - **Windows:** Run the .exe file and follow the installation wizard
 - **macOS:** Open the .dmg file and drag VS Code to Applications folder
 - **Linux:** Install using your package manager or download the .tar.gz archive
4. **First launch:** Open VS Code and you'll see the Welcome screen with helpful getting-started information

3.4. Essential Extensions for Python Programming

Extensions are what make VS Code incredibly powerful. Here are the must-have extensions for geospatial programming, organized by importance:

3.4.1. Core Extensions (Install First)

Python

- **What it does:** The foundation for Python development in VS Code
- **Key features:** Syntax highlighting, IntelliSense, debugging, linting, code formatting
- **Why it's essential:** Enables all Python functionality including code completion and error detection

Jupyter

- **What it does:** Native Jupyter notebook support within VS Code
- **Key features:** Run notebooks, interactive Python, data visualization
- **Why it's essential:** Perfect for geospatial data exploration and analysis

3.4.2. Development Enhancement Extensions

These extensions improve your coding experience and productivity:

Code Quality and Formatting:

- **Black Formatter:** Automatic Python code formatting following PEP 8 standards
- **Pylint:** Advanced Python linting for catching errors and enforcing coding standards
- **autoDocstring:** Automatically generates Python docstrings for your functions
- **Prettier:** Code formatter for JavaScript, JSON, CSS, and more

Development Tools:

- **Docker:** Docker support for VS Code

- **IntelliCode**: AI-assisted code completion and suggestions
- **CodeSnap**: Generate beautiful code screenshots for documentation

3.4.3. Git and Collaboration Extensions

Essential for version control and team collaboration:

Version Control:

- **GitLens**: Supercharge Git capabilities with blame annotations, history, and more
- **GitHub Pull Requests**: Manage GitHub pull requests directly in VS Code
- **GitHub Actions**: Edit and monitor GitHub Actions workflows

AI-Powered Development:

- **GitHub Copilot**: AI pair programmer that suggests code as you type
- **GitHub Copilot Chat**: Conversational AI for coding questions and explanations

3.4.4. Documentation Extensions

Important for sharing your geospatial work:

Documentation:

- **Markdown All in One**: Complete Markdown support with preview, shortcuts, and table of contents
- **Markdown Shortcuts**: Quick formatting shortcuts for Markdown documents

3.4.5. Installing Extensions

Method 1: Using the Extensions Panel

1. Click the Extensions icon in the Activity Bar (Figure 2)
2. Search for the extension name
3. Click “Install” on the extension you want

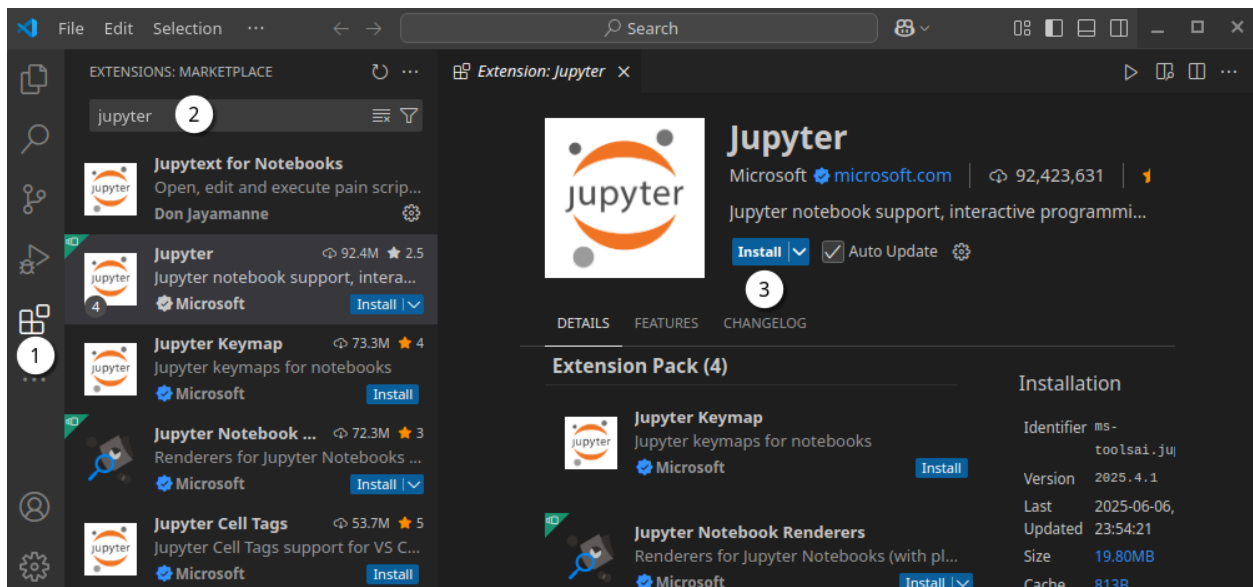


Figure 2: Install VS Code extensions from the Extensions panel.

Method 2: Using the Command Palette

1. Press **Ctrl+Shift+P** to open Command Palette
2. Type “Extensions: Install Extensions”
3. Search and install extensions

3.5. Configure VS Code for Python Development

In the previous chapter, we created a new conda environment called `geo` with Python 3.12 and installed the essential geospatial packages. Now, we need to set up VS Code to use this environment. First, create a new folder (e.g., `gispro`) and open it in VS Code. Next, go to **File > New File**. In the dialog that appears, enter a file name (e.g., `hello-world.py`) and select **Python File** (see [Figure 3](#)). This will create a new Python file in the current folder.

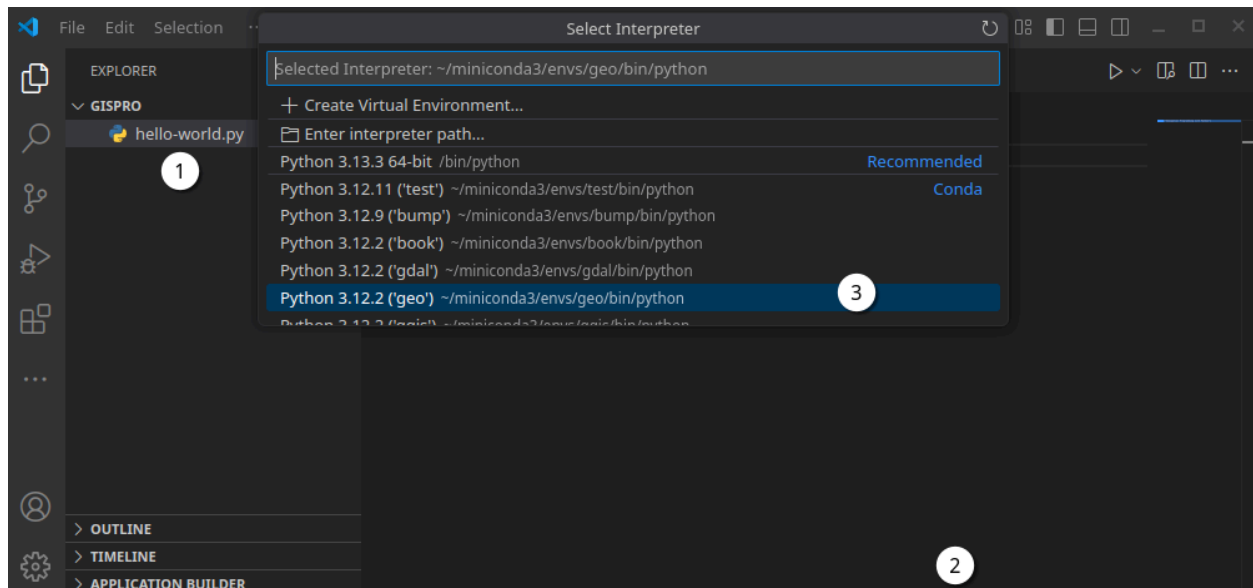


Figure 3: Select the Python interpreter for the current workspace.

Once the file is created, you can start writing Python code in it. For example, you can write the following code to print a message:

```
print("Geospatial Programming with Python")
```

To run the code, go to **Run > Run Without Debugging** (or press **Ctrl+F5**). You should see the message “Geospatial Programming with Python” printed in the terminal. If the terminal is not visible, you can open it by going to **View > Terminal**.

As mentioned above, VS Code has native support for Jupyter notebooks. You can create a new Jupyter notebook by going to **File > New File**. Enter a file name (e.g., `hello-world.ipynb`) and select **Jupyter Notebook**. This will create a new Jupyter notebook in the current folder. Follow the steps below to run the notebook (see [Figure 4](#)):

1. Double-click the notebook file under the **Explorer** panel to open it.
2. Click on the **Select Kernel** button in the top right corner of the notebook and select the `geo` environment you created in the previous chapter.

3. Click on the + **Cell** button to add a new cell.
4. Set the cell type to code by selecting **Python** in the lower right corner of the cell.
5. Write the following code in the cell and click on the **Run** button to the left of the cell.

```
print("Geospatial Programming with Python")
```

You should see the message “Geospatial Programming with Python” printed in the notebook. Press **Ctrl+S** to save the notebook.

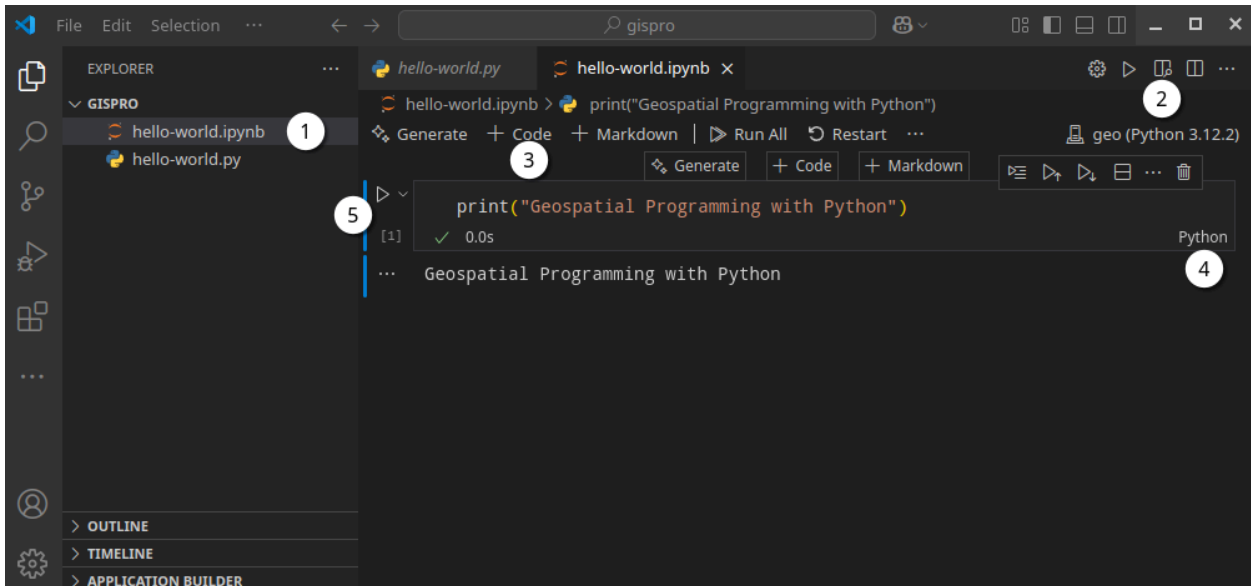


Figure 4: Run a Jupyter notebook in VS Code.

Congratulations! You have successfully set up VS Code for Python development.

3.6. Essential Keyboard Shortcuts

Learning keyboard shortcuts will dramatically improve your productivity. Here are the most important ones for geospatial programming:

3.6.1. Navigation and File Management

Core Navigation:

- **Command Palette:** **Ctrl+Shift+P** (Windows/Linux) or **Cmd+Shift+P** (macOS)
 - *Your gateway to all VS Code features*
- **Quick Open:** **Ctrl+P**
 - *Instantly open any file by typing its name*
- **Go to Symbol:** **Ctrl+Shift+O**
 - *Jump to functions, classes, or variables in current file*
- **Go to Line:** **Ctrl+G**
 - *Jump to specific line number*

File and Folder Operations:

- **New File:** Ctrl+N
- **Save:** Ctrl+S
- **Save All:** Ctrl+K S
- **Close Tab:** Ctrl+W
- **Reopen Closed Tab:** Ctrl+Shift+T

3.6.2. Code Editing and Manipulation

Text Selection and Editing:

- **Select Entire Line:** Ctrl+L or triple-click
- **Cut Entire Line:** Ctrl+X (when nothing is selected)
- **Copy Entire Line:** Ctrl+C (when nothing is selected)
- **Move Line Up/Down:** Alt+↑/↓
- **Duplicate Line:** Shift+Alt+↑/↓
- **Delete Line:** Ctrl+Shift+K

Multi-cursor Editing (powerful for data processing):

- **Add Cursor:** Alt+Click
- **Add Cursor Above/Below:** Ctrl+Alt+↑/↓
- **Select All Occurrences:** Ctrl+Shift+L
- **Select Next Occurrence:** Ctrl+D

3.6.3. Code Intelligence and Development

IntelliSense and Navigation:

- **Trigger IntelliSense:** Ctrl+Space
- **Go to Definition:** F12
- **Peek Definition:** Alt+F12
- **Go to References:** Shift+F12
- **Rename Symbol:** F2

Code Formatting:

- **Format Document:** Shift+Alt+F
- **Format Selection:** Ctrl+K Ctrl+F

3.6.4. Jupyter and Interactive Development

Jupyter Notebooks:

- **Run Cell:** Ctrl+Enter
- **Run Cell and Move to Next:** Shift+Enter
- **Insert Cell Above:** Ctrl+Shift+A
- **Insert Cell Below:** Ctrl+Shift+B
- **Delete Cell:** Ctrl+Shift+D

Python Interactive:

- **Run Selection in Terminal:** Shift+Enter
- **Run Python File:** Ctrl+F5

3.7. References and Further Learning

Official Documentation:

- [VS Code Python Tutorial](#)²³
- [Jupyter Notebooks in VS Code](#)²⁴

Keyboard Shortcuts Reference:

- [Windows Shortcuts](#)²⁵
- [macOS Shortcuts](#)²⁶
- [Linux Shortcuts](#)²⁷

3.8. Key Takeaways

Setting up VS Code effectively is crucial for productive geospatial programming. Here are the essential points to remember:

Core Setup:

- **VS Code + Python Extension** = Foundation for Python development
- **Jupyter Extension** = Interactive data exploration and analysis

Essential Workflows:

- Connect VS Code to your conda environments for consistent package management
- Use keyboard shortcuts to navigate and edit code efficiently
- Leverage integrated terminals for running scripts and managing environments
- Organize projects with clear folder structures and settings files

Productivity Features:

- **Multi-cursor editing** for processing multiple similar lines of data
- **IntelliSense and code completion** for faster, error-free coding
- **Integrated Git** for version control and collaboration
- **Debugging tools** for troubleshooting complex geospatial algorithms

Mastering VS Code will significantly improve your efficiency as a geospatial programmer. You'll spend less time fighting with tools and more time analyzing spatial data, creating visualizations, and solving real-world problems.

3.9. Exercises

These hands-on exercises will help you become proficient with VS Code for geospatial programming.

3.9.1. Exercise 1: Setting Up Your Geospatial Workspace

Objective: Configure VS Code for optimal geospatial development.

²³<https://code.visualstudio.com/docs/python/python-tutorial>

²⁴<https://code.visualstudio.com/docs/datascience/jupyter-notebooks>

²⁵<https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf>

²⁶<https://code.visualstudio.com/shortcuts/keyboard-shortcuts-macos.pdf>

²⁷<https://code.visualstudio.com/shortcuts/keyboard-shortcuts-linux.pdf>

Tasks:

1. Install VS Code and the essential extensions (Python, Jupyter, Remote Development)
2. Create a new folder called `gis-workspace`
3. Open the folder in VS Code (`File > Open Folder`)
4. Connect VS Code to your `geo` conda environment using “Python: Select Interpreter”
5. Create the recommended folder structure (data, notebooks, src, etc.)

3.9.2. Exercise 2: Mastering Essential Shortcuts

Objective: Develop muscle memory for the most important keyboard shortcuts.

Tasks:

1. Create a new Python file called `shortcut_practice.py`
2. Practice these specific shortcuts (use them 5 times each):
 - `Ctrl+P` to quickly open files
 - `Ctrl+Shift+P` to access command palette
 - `Alt+↑/↓` to move lines up and down
 - `Ctrl+D` to select multiple occurrences
 - `Alt+Click` to add cursors at different positions
3. Create a sample data processing script with multiple similar lines
4. Use multi-cursor editing to modify all similar lines simultaneously
5. Practice code formatting with `Shift+Alt+F`

3.9.3. Exercise 3: Jupyter Integration and Data Exploration

Objective: Use VS Code’s Jupyter capabilities.

Tasks:

1. Create a new Jupyter notebook in VS Code (`Ctrl+Shift+P` > “Create: New Jupyter Notebook”)
2. Create a new code cell and write some Python code to print a message.
3. Practice these Jupyter shortcuts:
 - `Shift+Enter` to run cells and move to next
 - `Ctrl+Enter` to run cells without moving
 - `Ctrl+Shift+A` and `Ctrl+Shift+B` to insert cells
4. Run the notebook and verify the message is printed.

Chapter 4. Version Control with Git

4.1. Introduction

4.1.1. What is Git?

Imagine you're working on a complex geospatial analysis project. You've spent weeks writing code to process satellite imagery, analyze land use patterns, and create beautiful visualizations. Then disaster strikes: you accidentally delete a crucial function, or an experiment goes wrong and corrupts your data processing pipeline. Without proper version control, you might lose days or weeks of work.

This is where **Git**²⁸ becomes your safety net and productivity superpower. Git is a distributed version control system that tracks every change to your files, creating a complete history of your project's evolution. Think of it as a sophisticated “undo” system that never forgets, combined with powerful collaboration tools that enable seamless teamwork.

Why Git is essential for geospatial programming:

- **Safety net:** Never lose work again—every version of your code is safely stored
- **Experimentation:** Try new approaches fearlessly, knowing you can always revert changes
- **Collaboration:** Work seamlessly with team members on complex geospatial projects
- **Documentation:** Track what changed, when, and why with detailed commit messages
- **Reproducibility:** Ensure others can recreate your exact analysis environment
- **Professional workflow:** Industry-standard tool used by developers worldwide

Git integrates seamlessly with the tools you've already learned—VS Code has excellent Git support, and Jupyter notebooks work beautifully with Git workflows. Combined with platforms like GitHub, Git provides the foundation for modern, collaborative geospatial development.

4.1.2. What is GitHub?

GitHub²⁹ is the world's largest platform for hosting Git repositories and collaborating on code. Think of it as a social network for developers, combined with powerful project management tools. While Git handles version control on your local machine, GitHub provides:

Key GitHub Features:

- **Remote hosting:** Store your repositories safely in the cloud
- **Collaboration:** Work with others through pull requests and code reviews
- **Backup:** Automatic backup of all your project versions
- **Sharing:** Make your code available to others or keep it private
- **Issue tracking:** Manage bugs and feature requests
- **Documentation:** Host project websites and documentation
- **Integration:** Connect with other development tools and services

GitHub vs. Git:

- **Git** = The version control system that runs on your computer
- **GitHub** = The online platform that hosts Git repositories and adds collaboration features

²⁸<https://git-scm.com>

²⁹<https://github.com>

4.2. Learning Objectives

By the end of this chapter, you should be able to:

- Create a GitHub account and understand GitHub fundamentals
- Install and configure Git for geospatial development workflows
- Create and manage local Git repositories for your geospatial projects
- Connect local repositories to GitHub and manage remote repositories
- Track changes, commit code, and write meaningful commit messages
- Use basic branching strategies for organizing your work
- Collaborate effectively using GitHub's pull request workflow
- Apply Git best practices specific to geospatial data and code
- Integrate Git workflows with VS Code and other development tools

4.3. Setting Up GitHub Account

Before installing Git, let's create your GitHub account since you'll use the same email address for both Git configuration and GitHub.

4.3.1. Creating Your GitHub Account

1. **Visit GitHub:** Go to <https://github.com>
2. **Sign up:** Click "Sign up" and provide:
 - **Username:** Choose carefully—this will be part of your repository URLs
 - **Email:** Use your professional or primary email address
 - **Password:** Create a strong, unique password
3. **Verify your account:** Check your email for verification instructions
4. **Choose your plan:**
 - **Free:** Unlimited public and private repositories (recommended for most users)
 - **Pro:** Additional features for advanced users

Important: Remember your GitHub email address—you'll use it when configuring Git.

4.4. Installing Git

Git is a free, open-source version control system available on all major platforms.

4.4.1. Installation Instructions

1. **Go to the official Git website:** <https://git-scm.com/downloads>
2. **Install Git based on your operating system:**
 - **Windows:** Download and run the installer `.exe` file
 - **macOS:** Use Homebrew: `brew install git`
 - **Linux (Debian/Ubuntu):** `sudo apt install git`

4.4.2. Verifying Installation

After installation, verify Git is working correctly:

```
# Check Git version
git --version

# Check installation location
which git # macOS/Linux
where git # Windows
```

You should see output showing the Git version number (e.g., `git version 2.49.0`).

4.5. Configuring Git

Configure Git with your identity using the same email address from your GitHub account.

4.5.1. Essential Configuration

Set your identity (required for all commits):

```
# Replace with your actual name and GitHub email
git config --global user.name "Your Full Name"
git config --global user.email "your.github@email.com"
```

4.5.2. Verification

Check your configuration:

```
# View all global configuration
git config --global --list

# Check specific values
git config --global user.name
git config --global user.email
```

Please make sure you use the same email address for Git and GitHub. Otherwise, you will not be able to push your changes to GitHub.

4.6. Understanding Git Concepts

Before diving into commands, let's understand the key concepts that make Git powerful for geospatial programming.

4.6.1. Core Git Concepts

Repository (Repo): A folder containing your project files and Git's tracking information

Commit: A snapshot of your project at a specific point in time

- Each commit has a unique ID and a descriptive message
- Like a save point—you can always return to this exact state

Branch: A parallel version of your repository

- `main` branch typically contains stable, working code
- Feature branches contain experimental or in-development work

Remote: A version of your repository stored on a server (like GitHub)

- Enables collaboration and serves as a backup

4.6.2. The Git Workflow

Git follows a simple workflow:

1. **Modify** files in your working directory
2. **Stage** changes you want to include in the next commit
3. **Commit** staged changes with a descriptive message
4. **Push** commits to GitHub (for sharing/backup)

```
Working Directory → Staging Area → Local Repository → GitHub
(edit)             (git add)       (git commit)   (git push)
```

4.7. Essential Git Commands

4.7.1. Starting a New Project

Create a new repository:

To create a new repository, you can use the following commands:

```
# Navigate to your project folder
cd ~/my-geo-project

# Initialize Git repository
git init

# Check repository status
git status
```

Clone an existing repository:

To clone an existing repository, you need to know the URL of the repository. You can find the URL of a repository by clicking the “Code” button on the GitHub repository page. For example, the URL of the `intro-gispro` repository is `https://github.com/giswqs/intro-gispro.git`.

Before cloning the repository, it is recommended to fork the repository to your own GitHub account by clicking the “Fork” button on the GitHub repository page. This will create a copy of the repository in your own GitHub account. Otherwise, you will not be able to push your changes to the original repository. Once you have forked the repository, you can clone it to your local machine using the following command:

```
# Clone a repository from GitHub
git clone https://github.com/<your-username>/intro-gispro.git
```

4.7.2. Tracking Changes

Add files to staging area:

Once you have cloned the repository, you can start to make changes to the files in the repository. For example, you can create a new file `analysis.py` in the root directory of the repository. Then you can add the file to the staging area using the following command:

```
# Add specific files
git add analysis.py
```

You can also add all files in the current directory to the staging area using the following command:

```
# Add all files in current directory
git add .
```

Check what's changed:

To check what's changed, you can use the following command:

```
# See status of all files
git status

# See detailed changes
git diff
```

Commit changes:

To commit changes, you can use the following command:

```
# Commit with inline message
git commit -m "Add rainfall analysis function"

# Commit all tracked files (skip staging)
git commit -am "Update visualization parameters"
```

4.7.3. Writing Good Commit Messages

Effective commit messages are crucial for geospatial projects. Here are some good examples:

```
git commit -m "Add NDVI calculation for Landsat 8 imagery"
git commit -m "Fix coordinate transformation bug in UTM conversion"
git commit -m "Update flood mapping algorithm to handle edge cases"
```

Format: Start with a verb, keep under 50 characters, be specific about what changed.

4.7.4. Working with GitHub

To push changes to GitHub, you first need to connect your local repository to the remote repository on GitHub. This is done by adding the remote repository URL to your local repository. You can do this by using the following command:

```
# Add remote repository
git remote add origin https://github.com/<your-username>/<repo-name>.git
```

Once you have connected your local repository to the remote repository on GitHub, you can push your changes to GitHub using the following command:

```
# Push to GitHub (first time)
git push -u origin main

# Push subsequent changes
git push
```

To pull changes from GitHub, you can use the following command:

```
# Pull changes from GitHub
git pull
```

4.7.5. Basic Branching

Create and switch branches:

Branching is a powerful feature of Git that allows you to work on multiple versions of your code in parallel. It is a good practice to create a new branch for each new feature or bug fix.

To create a new branch, you can use the following command:

```
# Create new branch for experimental feature
git checkout -b satellite-analysis

# List all branches
git branch

# Switch back to main branch
git checkout main

# Merge feature branch into main
git merge satellite-analysis
```

```
# Delete merged branch
git branch -d satellite-analysis
```

4.7.6. Viewing Project History

To view the history of the project, you can use the following command:

```
# View commit history
git log

# Compact one-line format
git log --oneline

# View specific file history
git log -- data_processing.py
```

4.7.7. Undoing Changes

If you want to undo the last commit, you can use the following command:

```
# Undo last commit but keep changes
git reset --soft HEAD~1
```

4.8. Using GitHub

GitHub is a powerful platform for collaboration and version control. It is a good practice to use GitHub to manage your geospatial projects. In fact, the code examples in this book are hosted on GitHub. You can find the repository at <https://github.com/giswqs/intro-gispro>. At the very least, it's essential to know how to clone a repository from GitHub to your local machine and pull updates to keep your local repository in sync with the latest changes from the remote version. Embracing these practices will enhance your workflow and project management.

4.8.1. Creating Repositories on GitHub

Two common workflows:

1. Start on GitHub, then clone:

- Create repository on GitHub with README
- Clone to your local machine: `git clone <url>`

2. Start locally, then push to GitHub:

- Create repository locally: `git init`
- Create matching repository on GitHub
- Connect and push: `git remote add origin <url>` then `git push -u origin main`

4.8.2. Basic Collaboration

Working with others:

1. **Fork** someone else's repository to your GitHub account
2. **Clone** your fork to your local machine
3. Make changes and **commit** them locally
4. **Push** changes to your fork on GitHub
5. Create a **Pull Request** to suggest changes to the original repository

4.8.3. Repository Management

Best practices:

- Use descriptive repository names (e.g., `landsat-ndvi-analysis`)
- Write clear README files explaining your project
- Use `.gitignore` files to exclude unnecessary files
- Make repositories public to share your work (or private for sensitive projects)

4.9. Integration with VS Code

VS Code provides excellent Git integration:

- **Source Control panel** (`Ctrl+Shift+G`): Visual interface for staging and committing
- **Built-in diff viewer**: See changes side-by-side
- **GitLens extension**: Enhanced Git capabilities with blame annotations
- **GitHub integration**: Create pull requests directly from VS Code

4.10. Best Practices for Geospatial Projects

4.10.1. Repository Structure

Organize your geospatial projects:

```
my-geo-project/
├── README.md
├── .gitignore
├── requirements.txt
├── data/
│   ├── raw/
│   └── processed/
├── src/
│   ├── data_processing.py
│   └── analysis.py
├── notebooks/
│   └── exploratory_analysis.ipynb
├── outputs/
│   ├── figures/
│   └── results/
```

4.10.2. What to Track vs. Ignore

Track these files:

- Source code (`.py` , `.r` , `.sql`)
- Jupyter notebooks (`.ipynb`)
- Configuration files (`requirements.txt` , `environment.yml`)
- Documentation (`.md` , `.rst`)
- Small reference datasets

Don't track these files (add to `.gitignore`):

- Large data files (use Git LFS or external storage)
- Temporary files (`.tmp` , `.cache`)
- Environment files (`.env` with secrets)
- Output files that can be regenerated
- IDE-specific files (`.vscode/settings.json`)

4.10.3. Commit Message Conventions

Use consistent format:

```
# Format: <type>: <description>
git commit -m "feat: add NDVI calculation for Sentinel-2"
git commit -m "fix: handle NoData values in elevation processing"
git commit -m "docs: update README with installation instructions"
```

Types: `feat` (new feature), `fix` (bug fix), `docs` (documentation), `refactor` (code cleanup)

4.11. Key Takeaways

Essential Git Workflow:

- `git init` or `git clone` = Start projects
- `git add` -> `git commit` -> `git push` = Save and share changes
- `git pull` = Get updates from GitHub
- `git status` and `git log` = Check current state and history

GitHub Integration:

- Remote repositories provide backup and collaboration
- Pull requests enable code review and contribution
- Issues and discussions facilitate project management
- GitHub Pages can host project documentation

Professional Practices:

- Write clear commit messages describing spatial analysis changes
- Use branches for experiments and new features
- Organize repositories with logical folder structures
- Use `.gitignore` to exclude large data files and temporary outputs
- Make code available for reproducible research

Collaboration Benefits:

- Share geospatial analysis methods with the research community
- Contribute to open-source geospatial libraries
- Build a portfolio showcasing your programming skills
- Work effectively with distributed teams

Mastering Git and GitHub will make you a more confident, collaborative, and professional geospatial programmer. You'll never lose work, can experiment fearlessly, and can contribute to the broader geospatial community.

4.12. Exercises

These exercises will help you master Git and GitHub for geospatial programming workflows.

4.12.1. Exercise 1: Setting Up Git and GitHub

Objective: Create accounts and configure your development environment.

Tasks:

1. Create a GitHub account if you don't have one
2. Install Git on your system
3. Configure Git with your name and GitHub email address
4. Verify your configuration is correct

Expected outcome: Properly configured Git ready to work with GitHub.

4.12.2. Exercise 2: Your First Repository

Objective: Create and manage a basic geospatial project repository.

Tasks:

1. Create a new repository on GitHub called `my-first-geo-project`
2. Clone it to your local machine
3. Create a basic folder structure (data/, src/, notebooks/)
4. Add a README.md file describing a simple geospatial analysis project
5. Create a `.gitignore` file appropriate for Python/geospatial work
6. Commit and push your changes to GitHub

Expected outcome: A properly structured repository on GitHub with initial project setup.

4.12.3. Exercise 3: Collaboration and Pull Requests

Objective: Practice GitHub collaboration workflow.

Tasks:

1. Fork a simple geospatial repository (find one on GitHub or use a provided example)
2. Clone your fork to your local machine
3. Create a new branch for your contribution
4. Make a small improvement (fix documentation, add comments, etc.)
5. Commit your changes and push the branch to your fork

6. Create a pull request back to the original repository
7. Write a clear description of your changes

Expected outcome: Understanding of the fork-clone-branch-PR workflow for collaboration.

Chapter 5. Using Google Colab

5.1. Introduction

Google Colab³⁰ is a cloud-based Jupyter notebook environment that has revolutionized how we approach geospatial programming and data science education. Think of it as having a powerful computer in the cloud that you can access from anywhere, with the ability to share your work instantly with colleagues around the world. It is free to use and requires no installation on your local machine. It even provides free GPU and TPU access for machine learning and deep learning tasks. You will be able to run our book code examples in Colab with no additional setup.

Why Google Colab is transformative for geospatial programming:

- **Zero setup:** No installation, configuration, or environment management—just open a browser and start coding
- **Free GPU/TPU access:** Process satellite imagery and run machine learning models on powerful hardware
- **Pre-installed packages:** Popular geospatial libraries like GDAL, geemap, GeoPandas, and ipyleaflet are readily available
- **Seamless sharing:** Collaborate on geospatial analyses as easily as sharing a Google Doc
- **Google Drive integration:** Store and access your datasets and notebooks in the cloud
- **No local storage limits:** Work with large geospatial datasets without filling up your hard drive

This chapter will teach you to leverage Colab’s full potential for geospatial programming. You’ll learn to work efficiently in this cloud environment and integrate it seamlessly with your broader geospatial programming toolkit.

5.2. Learning Objectives

By the end of this chapter, you should be able to:

- Access and navigate Google Colab for geospatial programming projects
- Set up and configure Colab notebooks for geospatial programming
- Install and manage geospatial Python packages in the Colab environment
- Import, store, and manage geospatial datasets using Google Drive integration
- Share and collaborate on geospatial notebooks with colleagues and the community
- Troubleshoot common issues when working with geospatial data in Colab
- Run code examples in this book in Colab

5.3. Getting Started with Google Colab

Google Colab requires no installation—just a web browser and a Google account. Let’s explore how to set up your first geospatial programming environment.

5.3.1. Accessing Colab

Starting your first notebook:

1. **Visit Google Colab:** <https://colab.research.google.com>

³⁰<https://colab.research.google.com>

2. **Sign in** with your Google account
3. **Create a new notebook:** Click “New notebook” or use the scratchpad
4. **Choose your runtime:** By default, Colab provides a CPU runtime. You can change it to a GPU or TPU runtime if needed for your geospatial computations. Go to **Runtime** -> **Change runtime type** -> select **GPU** or **TPU**.

5.3.2. Understanding the Colab Interface

The Google Colab interface is shown in [Figure 5](#). It is a Jupyter notebook environment that allows you to write and execute Python code for spatial analysis. On the left side of the interface, you can see the file panel, which allows you to manage your files and folders. On the right side of the interface, you can see the code cells and text cells. You can write code in the code cells and execute them by clicking the **Run** button or using the keyboard shortcut **Ctrl + Enter**. You can also write text in the text cells and execute them by clicking the **Run** button or using the keyboard shortcut **Ctrl + Enter**.

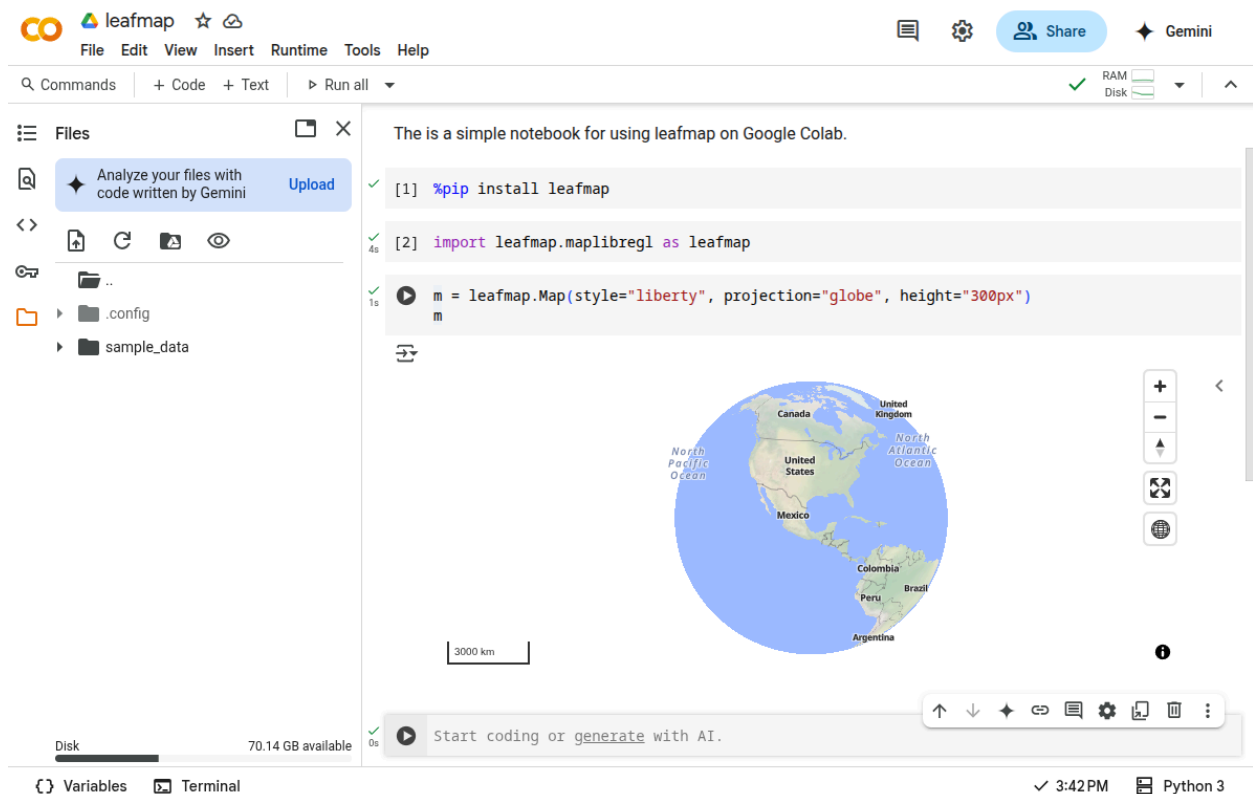


Figure 5: The Google Colab interface.

5.4. Setting Up Your Geospatial Environment

To use Colab for geospatial programming, you need to set up your environment first. First, check the pre-installed packages in Colab by running the following code:

```
%pip list
```

You should see a list of packages that are pre-installed in Colab. Some commonly used packages for geospatial programming that are pre-installed in Colab include: `gdal`, `geemap`, `geopandas`, and `xarray`. You can also install additional packages by running the following code:

```
%pip install <package-name>
```

To install multiple packages at once, you can use the following code:

```
%pip install <package-name1> <package-name2> <package-name3>
```

For example, to install the `leafmap` and `pygis` packages, you can run the following code:

```
%pip install leafmap pygis
```

Note that the `pip` command is prefixed with `%` in Colab. This is because Colab uses Jupyter notebooks, and the `%` is a special character in Jupyter notebooks called magic commands that allows you to run system commands. You can also use the `!` prefix to run system commands. For example, to install the `pygis` package, you can run the following code:

```
!pip install pygis
```

Sometimes, the installation log output might be overwhelming. You can suppress the output by adding `-q` to the end of the command. For example, to install the `pygis` package quietly, you can run the following code:

```
!pip install -q pygis
```

To check if the package is installed, you can run the following code:

```
%pip show geemap
```

You should see the information about the `geemap` package in the output, including the version number, package location, and other information. This is helpful if you want to know where the package is installed. At the time of writing, the Python version pre-installed in Colab is 3.11. All Python packages are installed in the `/usr/local/lib/python3.11/dist-packages` directory.

To uninstall a package, you can run the following code:

```
%pip uninstall geemap -y
```

The `-y` flag is used to automatically answer yes to the prompt.

5.5. Essential Colab Features

5.5.1. Code Execution and Navigation

To become proficient in using Colab, you need to know how to navigate the interface and execute code efficiently. The most commonly used keyboard shortcuts are shown below.

Keyboard shortcuts for efficient coding:

- **Run current cell:** Ctrl + Enter
- **Run cell and move to next:** Shift + Enter
- **Run cell and insert new cell below:** Alt + Enter
- **Run selected text:** Ctrl + Shift + Enter
- **Comment/uncomment:** Ctrl + /

Cell management shortcuts:

- **Insert cell above:** Ctrl + M + A
- **Insert cell below:** Ctrl + M + B
- **Delete cell:** Ctrl + M + D
- **Change to code cell:** Ctrl + M + Y
- **Change to markdown cell:** Ctrl + M + M

Navigation helpers:

- **Jump to class definition:** Ctrl + Click on class names
- **View function documentation:** Hover over functions
- **Code completion:** Tab key while typing

5.5.2. Working with Files and Data

Google Colab notebooks are temporary and will be deleted after you close the browser tab. To save your work, you need to save your notebook to your Google Drive. You can do this by clicking the **File** menu and selecting **Save a copy in Drive**. You can also save your notebook to your local machine by clicking the **File** menu and selecting **Download .ipynb**.

5.5.2.1. Mount Google Drive for Persistent Storage

To access your Google Drive in Colab, you need to mount it first. You can do this by running the following code:

```
from google.colab import drive
drive.mount('/content/gdrive')

# Access your Google Drive and Shared drives
import os
os.listdir('/content/gdrive/MyDrive')
```

5.5.2.2. Upload Files from Your Computer

To upload files from your computer to Colab, you can use the following code:

```
from google.colab import files

# Upload single or multiple files
uploaded = files.upload()

# Upload and unzip data
!unzip sample_data.zip
```

5.5.2.3. Download Processed Results

To download files from Colab to your computer, you can use the following code:

```
from google.colab import files

# Download a single file
files.download('analysis_results.csv')

# Zip and download multiple files
!zip -r results.zip output_folder/
files.download('results.zip')
```

5.5.2.4. Access Web Datasets Directly

To download datasets from the web, you can use the following code:

```
# Download datasets from URLs
!wget https://opengeos.org/data/world/world_cities.csv

# Clone repositories with sample data
!git clone https://github.com/giswqs/intro-gispro.git
```

5.6. Run Code Examples in Colab

The code examples in this book can be run in Colab with no additional setup by clicking the [Open in Colab](#) badge on the repository page. Alternatively, you can follow the steps below to run the code examples in Colab:

1. Go to <https://colab.research.google.com/github/giswqs/intro-gispro/blob/main>
2. Select the notebook you want to run.

5.7. Key Takeaways

Google Colab revolutionizes geospatial programming by providing free access to powerful computing resources. Mastering Google Colab will democratize your access to high-performance geospatial computing, enable seamless collaboration, and accelerate your spatial analysis workflows. Whether you're a student learning GIS programming or a researcher processing global datasets, Colab provides the tools and resources you need.

5.8. Exercises

These hands-on exercises will help you master Google Colab for geospatial programming workflows.

5.8.1. Exercise 1: Setting Up Your Geospatial Colab Environment

Objective: Configure Google Colab for geospatial development.

Tasks:

1. Open Google Colab and create a new notebook
2. Check system resources (RAM, GPU availability, disk space)
3. Mount your Google Drive for persistent storage
4. Install essential geospatial packages (geopandas, rasterio, leafmap, maplibre)
5. Create a working directory structure in your Drive
6. Test the installation by creating a simple map with leafmap
7. Save your notebook with a descriptive name

5.8.2. Exercise 2: Data Management and File Operations

Objective: Master data handling workflows in the cloud environment.

Tasks:

1. Download a sample geospatial dataset using wget
2. Upload a file from your local computer to Colab
3. Unzip and organize the data in your Drive folder
4. Load the data into a Pandas Dataframe or a GeoPandas GeoDataFrame
5. Create a simple map with the data
6. Save the map as an image file
7. Download the image file to your local computer

5.8.3. Exercise 3: Collaborative Notebook Development

Objective: Create and share a comprehensive geospatial analysis notebook.

Tasks:

1. Create a sample notebook to create a simple map with leafmap
2. Add markdown cells explaining each step
3. Change the share settings to `Anyone with the link`
4. Share the notebook with your friends and ask them to run the notebook

Chapter 6. Working with JupyterLab

6.1. Introduction

JupyterLab³¹ is a powerful, web-based interactive development environment that serves as the next-generation interface for Jupyter notebooks. Think of it as an integrated workspace where you can write code, visualize data, manage files, and document your analysis—all within a single browser window. Unlike traditional code editors that separate different tools, JupyterLab brings everything together in one cohesive environment.

At its core, JupyterLab builds upon the familiar Jupyter notebook concept, where you can combine executable code, rich text, equations, and visualizations in a single document. However, JupyterLab extends this concept by providing a more flexible and feature-rich interface that accommodates complex workflows and project management needs.

JupyterLab is particularly well-suited for geospatial programming because spatial data analysis is inherently exploratory and iterative. When working with geographic data, you often need to:

- Load and inspect datasets from various sources
- Experiment with different processing methods
- Create and refine visualizations
- Document your methodology for reproducibility
- Share results with collaborators

In this chapter, we will learn how to install and configure JupyterLab for geospatial programming workflows. We will also learn how to run the code examples in this book in JupyterLab.

6.2. Learning Objectives

By the end of this chapter, you should be able to:

- Install and configure JupyterLab for geospatial programming workflows
- Navigate the JupyterLab interface efficiently for data analysis and visualization
- Create and manage notebooks, terminals, and files in a unified workspace
- Use keyboard shortcuts and productivity features for efficient coding

6.3. Installing and Setting Up JupyterLab

Getting JupyterLab up and running is the first step toward creating an effective geospatial programming environment. The installation process is straightforward. You can use either `conda` or `pip` to install JupyterLab.

6.3.1. Using `conda`

This is the preferred method for most geospatial programming scenarios. It is recommended to create a new `conda` environment for geospatial programming.

³¹<https://jupyter.org>

```
conda create -n geo python=3.12
conda activate geo
conda install -c conda-forge jupyterlab leafmap
```

6.3.2. Using pip

If you are using a pip-based environment, you can install JupyterLab using pip.

```
pip install jupyterlab leafmap
```

6.3.3. Verifying Your Installation

After installation, it's important to verify that everything is working correctly. This step helps catch any issues before you start working on important projects.

To verify your installation, you can run the following command in your terminal:

```
jupyter lab --version
```

You should see the version number of JupyterLab. At the time of writing, the latest version of JupyterLab is 4.4.1. If you see an error message, check that your environment is activated and packages installed correctly.

6.4. Getting Started with JupyterLab

Once JupyterLab is installed, the next step is learning how to launch it effectively and navigate its interface. Understanding these basics will set you up for success in your geospatial programming journey.

6.4.1. Launching JupyterLab

JupyterLab runs as a web application, which means it operates through your web browser even though it's running locally on your computer. This design provides the flexibility of a web interface with the power of local computing.

To launch JupyterLab, navigate to the directory where you want to work and run the following command in your terminal:

```
jupyter lab
```

You should see the JupyterLab interface in your browser (see [Figure 6](#)). By default, JupyterLab will launch on port 8888.

You can also launch JupyterLab on a specific port by adding the `--port` option. For example, to launch JupyterLab on port 8889, you can run the following command:


```
# Launch on a specific port (useful if the default port is busy)
jupyter lab --port=8889

# Launch without automatically opening browser (useful for remote servers)
jupyter lab --no-browser --port=8888
```

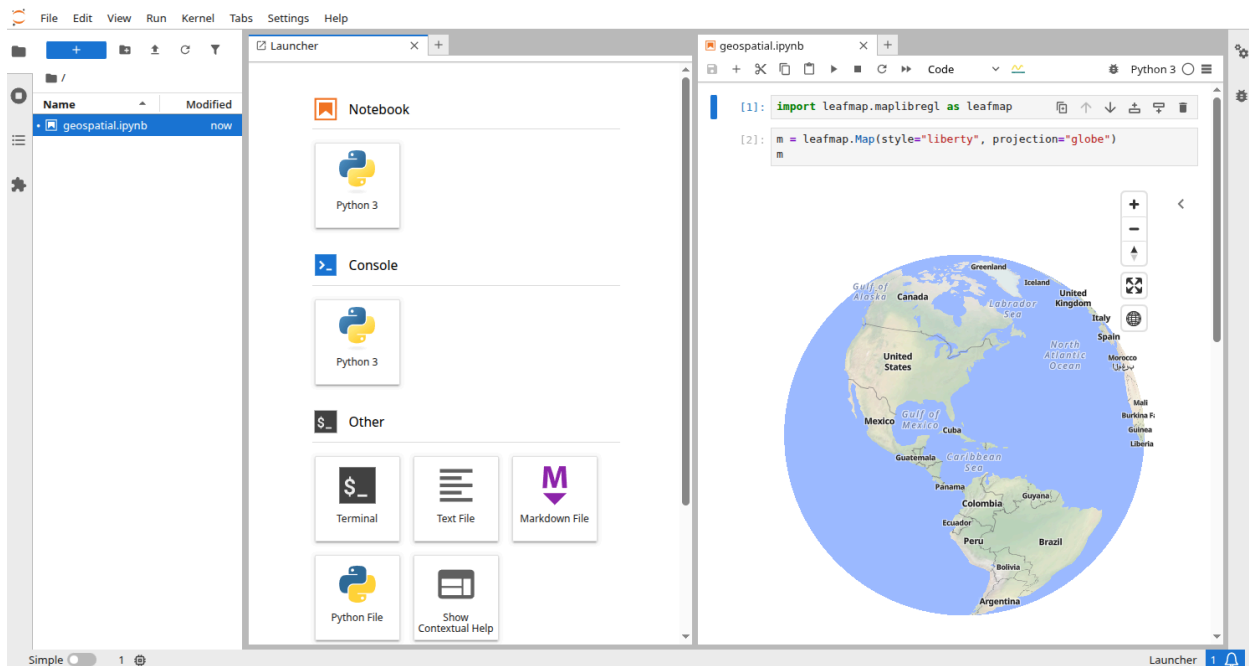


Figure 6: The JupyterLab interface.

What happens when you launch JupyterLab:

1. **Server starts:** JupyterLab starts a local web server on your computer
2. **Browser opens:** Your default web browser opens to the JupyterLab interface
3. **Connection established:** The browser connects to the local server
4. **Interface loads:** You see the JupyterLab workspace

Important notes:

- Keep the terminal window open while using JupyterLab—closing it will shut down the server
- The URL will typically be `http://localhost:8888/lab` (or different port if specified)
- You can open multiple browser tabs to the same JupyterLab session

6.4.2. Understanding the JupyterLab Interface

JupyterLab's interface is designed around the concept of a flexible workspace that can adapt to different workflows.

The JupyterLab interface consists of four main areas:

1. Left Sidebar (Your Command Center)

The left sidebar provides access to different tools and views:

- **File Browser** (folder icon): Navigate your project files, create new folders, upload data files, and manage your project structure
- **Running Kernels** (circle icon): Monitor active notebooks and terminals, useful for managing multiple analyses
- **Table of Contents** (list icon): Navigate long notebooks by their headings
- **Extension Manager**: Add new functionality to JupyterLab

2. Main Work Area (Your Canvas)

This is where your actual work happens:

- **Tabbed interface**: Each notebook, text file, terminal, or data viewer opens in its own tab
- **Split views**: Drag tabs to create side-by-side or top-bottom layouts
- **Flexible arrangement**: Organize your workspace to match your workflow

3. Top Menu Bar (Your Toolbox)

Contains traditional menu items organized by function:

- **File**: Create, open, save, and export files
- **Edit**: Copy, paste, find, and replace operations
- **View**: Control interface layout and appearance
- **Run**: Execute notebook cells and manage kernels
- **Kernel**: Restart, interrupt, or change computational kernels
- **Tabs**: Switch between different notebooks, text files, terminals, and data viewers
- **Settings**: Change the appearance of the interface
- **Help**: Access documentation and keyboard shortcuts

4. Status Bar (Your Information Panel)

Shows important real-time information:

- **Kernel status**: Whether your Python environment is busy or idle
- **Line/column position**: Your current location in code
- **File information**: Details about the current file
- **System status**: Memory usage and other system information

6.4.3. Creating Your First Notebook in JupyterLab

In the **Launcher** tab, you can create a new notebook by clicking the “Python 3” button (see [Figure 6](#)). A new notebook will be created in a new tab. Type the following code in the first cell:

```
import leafmap.maplibregl as leafmap
```

Click the “Run” button to run the code. Another code cell will be created automatically. Type the following code in the second cell:

```
m = leafmap.Map(style="liberty", projection="globe")
m
```

Click the “Run” button to run the code. A map will be created in the third cell (see [Figure 6](#)). You can zoom in and out the map by scrolling the mouse wheel. We will learn more about `leafmap` in Section III of this book about geospatial programming with Python.

Once you are done with the notebook, you can save it by clicking the **File** menu and selecting **Save Notebook**. Alternatively, you can press `Ctrl+S` to save the notebook.

6.5. Essential Keyboard Shortcuts

Mastering keyboard shortcuts is one of the most effective ways to increase your productivity in JupyterLab. While it might seem overwhelming at first, learning these shortcuts will dramatically speed up your geospatial programming workflow. The key is to start with the most essential shortcuts and gradually build your muscle memory.

6.5.1. Understanding Notebook Modes: The Foundation

Before learning specific shortcuts, it's crucial to understand that JupyterLab notebooks operate in two distinct modes. Understanding these modes is essential because many shortcuts work differently depending on which mode you're in.

Edit Mode

- **When you're in it:** When you're actively typing or editing the content within a cell
- **What it looks like:** The cell has a blue border, and you can see a cursor inside the cell
- **What you can do:** Type code or text, select and edit content within the cell
- **How to enter:** Press `Enter` when a cell is selected, or click inside a cell
- **Think of it as:** "Content editing mode"

Command Mode

- **When you're in it:** When you're navigating between cells or working with the notebook structure
- **What it looks like:** There is a blue border larger than the cell, and you can see the entire cell is selected
- **What you can do:** Create, delete, move, and modify cells; navigate between cells; change cell types
- **How to enter:** Press `Esc` from edit mode, or click outside the cell content area
- **Think of it as:** "Notebook management mode"

6.5.2. The Most Important Shortcuts

These are the shortcuts you'll use constantly in geospatial programming. Master these before moving on to others:

Universal shortcuts (work in both modes):

- `Shift + Enter` : **Run current cell and move to next**
 - *This is the most important shortcut!* You'll use it constantly to execute code and see results
 - Perfect for step-by-step analysis of geospatial data
- `Ctrl + Enter` : **Run current cell and stay in place**
 - Useful when you want to re-run a cell multiple times (like adjusting a map visualization)
- `Alt + Enter` : **Run current cell and insert new cell below**
 - Great for iterative analysis—run your code and immediately get a new cell for the next step
- `Ctrl + S` : **Save notebook**
 - Critical for preserving your work, especially before long computations

6.5.3. Command Mode Shortcuts (Press `Esc` first)

Once you're comfortable with the universal shortcuts, these command mode shortcuts will significantly speed up your workflow:

Creating and Managing Cells:

- **A : Insert cell above current cell**
 - Useful when you need to add documentation or setup code before existing analysis
- **B : Insert cell below current cell**
 - Most common way to add new cells as you build your analysis
- **DD : Delete selected cell** (press D twice)
 - The double-press prevents accidental deletions
 - Don't worry—you can undo with `Z`, not `Ctrl+Z`
- **Z : Undo cell deletion**
 - Recovers accidentally deleted cells

Copying and Moving Cells:

- **C : Copy selected cell(s)**
- **X : Cut selected cell(s)**
- **V : Paste cell(s) below**
 - Useful for duplicating analysis steps or reorganizing your notebook

Changing Cell Types:

- **Y : Change cell to Code**
 - Convert markdown cells back to code cells
- **M : Change cell to Markdown**
 - Convert code cells to documentation cells
 - Essential for creating well-documented geospatial analyses

Navigation and Selection:

- **↑/↓ : Navigate between cells**
 - Much faster than clicking, especially in long notebooks
- **Shift + ↑/↓ : Extend cell selection**
 - Select multiple cells for batch operations
- **Shift + M : Merge selected cells**
 - Combine multiple cells into one (useful for consolidating code)

6.5.4. Edit Mode Shortcuts (Press `Enter` first)

These shortcuts help you write and edit code more efficiently:

Code Assistance:

- **Tab : Code completion or indent**
 - Essential for exploring geospatial library functions and methods
 - Example: Type `leafmap.` and press `Tab` to see available `leafmap` methods
- **Shift + Tab : Show function documentation**
 - Place the cursor on a function and press `Shift + Tab` to see the documentation of the function

Text Editing:

- **Ctrl + / : Toggle line comment**

- Quickly comment/uncomment lines of code for testing
- **Ctrl + A : Select all text in cell**
- **Ctrl + Z : Undo text edit**
 - Different from command mode undo—this undoes changes within a cell

6.6. Running Code Examples on MyBinder

[MyBinder](https://mybinder.org)³² is a free service that allows you to run code examples in JupyterLab without having to install anything on your computer. You can use MyBinder to run the code examples in this book. On the GitHub page of this book, you can click the **Launch Binder** button to launch the code examples in JupyterLab. Alternatively, you can use the following URL directly:

<https://mybinder.org/v2/gh/giswqs/intro-gispro/HEAD>

It may take a few minutes to launch the JupyterLab environment. Once the environment is ready, you can navigate to the `book` folder and open any notebook to run it.

6.7. Key Takeaways

JupyterLab serves as the foundation for modern geospatial programming, providing an integrated environment that brings together all the tools you need for spatial data analysis. By mastering the fundamentals covered in this chapter, you’ve built a solid foundation for your geospatial programming journey.

What you’ve learned:

- **Installation and setup:** How to properly install JupyterLab with geospatial packages using conda for maximum compatibility
- **Interface navigation:** Understanding the four main areas of JupyterLab and how they work together to support geospatial workflows
- **Notebook creation:** Best practices for organizing geospatial analyses with clear structure and documentation
- **Essential shortcuts:** The key keyboard shortcuts that will dramatically improve your productivity
- **Code examples:** How to run the code examples in this book in JupyterLab environment on MyBinder

6.8. Exercises

These exercises are designed to help you build practical skills with JupyterLab in a geospatial context. Work through them at your own pace, and don’t hesitate to experiment beyond the basic requirements.

6.8.1. Exercise 1: Setting Up Your Geospatial JupyterLab Environment

Objective: Install and configure JupyterLab for geospatial programming workflows.

What you’ll learn: Environment management, package installation, and basic configuration.

Step-by-step tasks:

1. **Create a dedicated geospatial environment:**

³²<https://mybinder.org>

```
conda create -n geolab python=3.12
conda activate geolab
```

1. Install JupyterLab and essential geospatial packages:

```
conda install -c conda-forge jupyterlab geopandas matplotlib ipyleaflet
```

1. Launch and explore JupyterLab:

- Start JupyterLab: `jupyter lab`
- Explore each area of the interface (sidebar, main area, menu bar, status bar)
- Open multiple tabs and practice arranging them

2. Test your installation:

- Create a new notebook
- Test basic imports:

```
import geopandas as gpd
import matplotlib.pyplot as plt
import ipyleaflet
print("Geospatial environment ready!")
```

6.8.2. Exercise 2: Mastering Keyboard Shortcuts and Efficient Workflow

Objective: Develop fluency with JupyterLab navigation and editing shortcuts.

What you'll learn: Efficient notebook editing, keyboard shortcuts, and productivity techniques.

Step-by-step tasks:

1. Practice basic shortcuts (spend 10 minutes on each):

- Cell execution: `Shift + Enter`, `Ctrl + Enter`, `Alt + Enter`
- Mode switching: `Esc` and `Enter`
- Cell creation: `A` and `B` (in command mode)
- Cell deletion: `DD` and recovery with `Z`

2. Create a practice notebook using only shortcuts:

- Create 8 cells of mixed types (code and markdown)
- Practice changing cell types: `M` for markdown, `Y` for code
- Navigate using arrow keys in command mode
- Practice cell copying: `C`, `V`, and `X`

3. Test code assistance shortcuts:

- Type `world.` and press `Tab` to see methods
- Use `Shift + Tab` on functions like `gpd.read_file()` to see documentation
- Practice commenting code with `Ctrl + /`

4. Workflow efficiency challenges:

- Create a new notebook and add 5 cells using only shortcuts
- Rearrange cells by copying and pasting

- Convert all cells to markdown, then back to code
- Delete and recover cells using `DD` and `Z`

5. **Time yourself:**

- See how quickly you can create a 10-cell notebook with alternating code/markdown cells
- Practice until you can do this in under 2 minutes

Chapter 7. Using Docker

7.1. Introduction

Have you ever tried to install geospatial python packages on a new computer, only to run into compatibility issues, missing dependencies, or version conflicts? Or maybe you've had code that works perfectly on your machine but fails when a colleague tries to run it on theirs? These are common problems in geospatial programming, where we often work with complex libraries that have many dependencies.

Docker³³ solves these problems by providing a way to package your entire development environment – including the operating system, Python, libraries, and your code – into a **container**. Think of a container like a lightweight virtual computer that runs inside your actual computer. This container includes everything needed to run your geospatial programs, and it works the same way on any computer that has Docker installed.

7.1.1. Why Use Docker for Geospatial Programming?

Docker offers several key advantages for geospatial work:

- **Consistency:** Your code runs the same way on your laptop, your colleague's computer, and on cloud servers. No more "it works on my machine" problems.
- **Easy Setup:** Instead of spending hours installing GDAL, GEOS, PROJ, Rasterio, and other geospatial libraries, you can start with a pre-configured environment in minutes.
- **Isolation:** Each project can have its own environment without conflicts. You can work on one project that uses Python 3.10 and another that uses Python 3.13, without any issues.
- **Reproducibility:** Your research becomes more reproducible when others can run your exact environment.
- **Portability:** Move your work between your laptop, a university server, or cloud platforms without reinstalling anything.

7.2. Learning Objectives

By the end of this chapter, you will be able to:

- Understand what Docker is and why it's useful for geospatial programming
- Install Docker on your computer
- Understand basic Docker concepts like containers and images
- Use Docker to work with Jupyter notebooks and geospatial libraries
- Run code examples in this book in Docker
- Understand the benefits of containerized development environments

7.3. Installing Docker Desktop

Docker Desktop is the easiest way to get started with Docker on Windows, macOS, and Linux. Below are the installation instructions for Windows, macOS, and Linux.

³³<https://www.docker.com>

7.3.1. Windows and macOS

1. Go to <https://www.docker.com/products/docker-desktop>
2. Download Docker Desktop for your operating system
3. Run the installer and follow the setup instructions
4. After installation, Docker Desktop will start automatically
5. You'll see the Docker icon in your system tray (Windows) or menu bar (macOS)

7.3.2. Linux

There are multiple ways to install Docker on Linux, depending on the distribution. Please refer to the [Docker installation guide](#)³⁴ for the latest instructions.

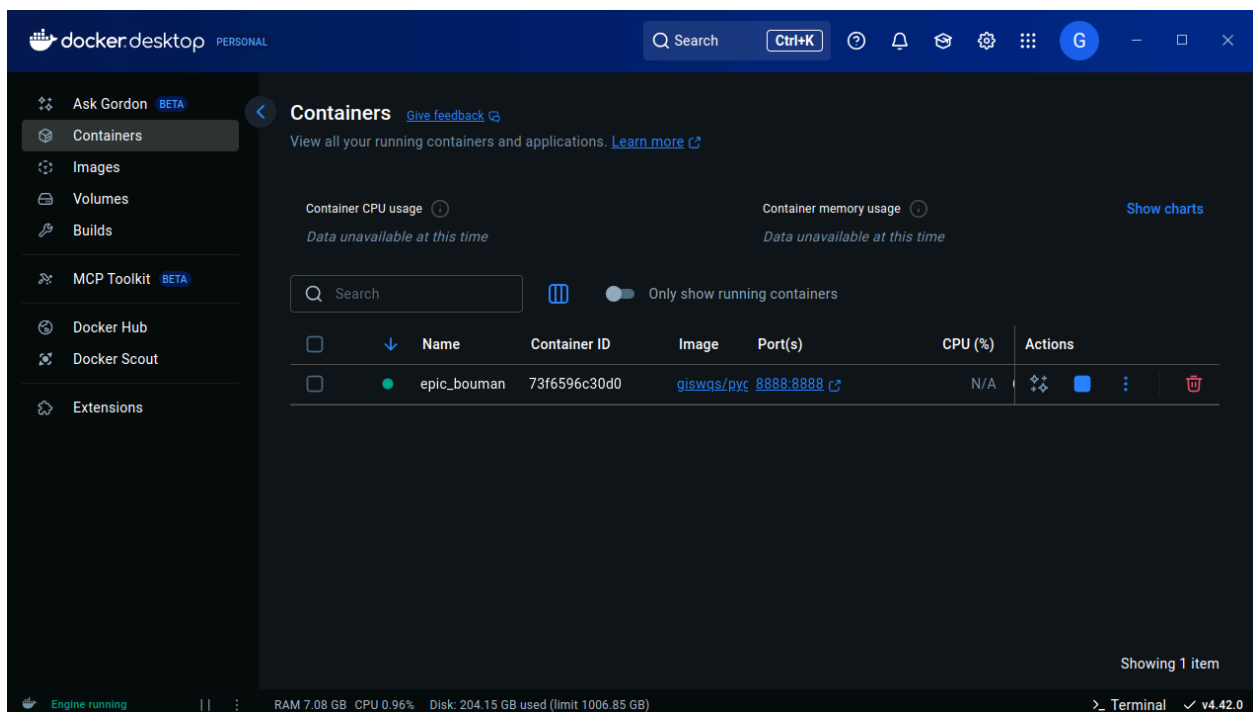
7.3.3. Verifying Installation

To check if Docker is installed correctly, open a terminal or command prompt and run:

```
docker --version
```

You should see output showing the Docker version. At the time of writing, the latest version of Docker Desktop is 4.42.0.

To start Docker Desktop ([Figure 7](#)), click the Docker icon in the system tray (Windows) or menu bar (macOS). You can manage Docker containers and images from Docker Desktop. Alternatively, you can manage Docker containers and images from the terminal, which will be covered in the next section.



³⁴<https://docs.docker.com/desktop/setup/install/linux/>

Figure 7: The Docker Desktop application.

7.4. Basic Concepts

Understanding a few basic concepts will help you use Docker effectively for geospatial programming.

7.4.1. Images vs. Containers

Docker Image: Think of an image as a blueprint or template. It contains the operating system, Python, all the libraries, and sometimes your code. Images are read-only and can be shared between computers.

Docker Container: A container is a running instance of an image. It's like creating a virtual computer from the blueprint. You can start, stop, and interact with containers.

Analogy: If an image is like a cookie cutter, then a container is like the actual cookie you make with it.

7.4.2. Key Docker Terms

Dockerfile: A text file that contains instructions for building a Docker image (we won't create these in this chapter, but you might see them referenced).

Docker Hub: An online repository where people share Docker images. Think of it like an app store for Docker images.

Volume: A way to share files between your computer and a Docker container. This lets you edit files on your computer and have them available inside the container.

Port: Containers run in isolation, but you can “expose” ports to access services running inside them, like Jupyter notebooks.

7.5. Running Code Examples in Docker

In the previous chapter, we learned how to run code examples on MyBinder. Now let's see how to run code examples in Docker.

The Docker image for this book is `giswqs/pygis:book`. We can pull it from Docker Hub by running:

```
docker pull giswqs/pygis:book
```

Then we can run the container by running:

```
docker run -it -p 8888:8888 -v $(pwd):/app/workspace giswqs/pygis:book
```

Let's break down this command:

- `docker run` : Start a new container
- `-it` : Make the container interactive (you can type commands into it)
- `-p 8888:8888` : Connect port 8888 inside the container to port 8888 on your computer (for Jupyter)
- `-v $(pwd):/app/workspace` : Share your current directory with the container
- `giswqs/pygis:book` : The name of the Docker image and the tag to use

After running this command, you should see output indicating that Jupyter is starting. Look for a line that shows:

```
http://127.0.0.1:8888/lab?token=your_token_here
```

Press **Ctrl** and click the link to open JupyterLab in your web browser. Alternatively, copy this URL into your web browser to access JupyterLab. Once you are in JupyterLab, you can navigate to the `book` folder and open any notebook to run the code examples.

Note that the `pygis` docker image has multiple tags, including `book` and `latest`. You can check the available tags by visiting the [Docker Hub page](https://hub.docker.com/r/giswqs/pygis/tags)³⁵.

You can use the `book` tag to run the code examples in this book. You can also use the `latest` tag to run the latest version of the `pygis` docker image. All the Python packages needed to run the code examples in this book are installed in the `pygis` docker image. You can install additional packages by running `pip install <package_name>` in the container.

7.5.1. Working with Files

When you use the `-v $(pwd):/app/workspace` option, your current directory on your computer is shared with the container. This means:

- Files you create in the container's `/app/workspace` folder will appear on your computer
- Files you create on your computer will appear in the container
- Your work is saved on your computer, not inside the container

7.5.2. Stopping a Container

To stop a running container, you can press `Ctrl+C` in the terminal where you started it

7.5.3. Listing Running Containers

To see what containers are currently running:

```
docker ps
```

7.6. Common Docker Commands

Here are the essential Docker commands you'll use most often:

7.6.1. Basic Operations

```
# Pull an image from Docker Hub
docker pull giswqs/pygis:book

# List downloaded images
```

³⁵<https://hub.docker.com/r/giswqs/pygis/tags>

```
docker images

# List running containers
docker ps

# List all containers (including stopped ones)
docker ps -a

# Stop a container
docker stop container_name

# Remove a container
docker rm container_name

# Remove an image
docker rmi image_name
```

7.6.2. Getting Help

```
# Get help for any Docker command
docker help
docker run --help
```

7.6.3. Choosing Port Numbers

If port 8888 is already in use on your computer, you can use a different port:

```
# Use port 8889 instead
docker run -it -p 8889:8888 -v $(pwd):/app/workspace giswqs/pygis:book
```

Then access Jupyter at `http://127.0.0.1:8889/lab?token=your_token_here` instead.

7.6.4. Saving Your Work

Remember that containers are temporary. Always make sure your important files are saved in the shared volume (the folder you mounted with `-v`) so they persist on your computer.

7.7. Key Takeaways

Docker provides a powerful solution for the common challenges in geospatial programming:

1. **Consistency:** Docker containers run the same way on any computer with Docker installed
2. **Easy Setup:** Pre-built images eliminate the need to manually install complex geospatial libraries
3. **Isolation:** Each project can have its own environment without conflicts
4. **Reproducibility:** Others can run your exact environment using the same Docker image

The basic workflow is:

1. Choose an appropriate Docker image (like `giswqs/pygis:book`)
2. Run the container with volume mounting to share files
3. Work in the containerized environment (often through Jupyter)
4. Your files are automatically saved to your computer through the volume mount
5. Stop the container when done

Docker bridges the gap between the complexity of geospatial software and the need for accessible, reproducible research environments.

7.8. Exercises

7.8.1. Exercise 1: First Docker Run

1. Install Docker on your computer following the instructions in this chapter
2. Create a new folder called `docker_test`
3. Navigate to this folder in your terminal
4. Run the leafmap Docker container:

```
docker run -it -p 8888:8888 -v $(pwd):/home/jovyan/work ghcr.io/opegeos/leafmap:latest
```

5. Open the JupyterLab URL in your browser
6. Navigate to the `leafmap/docs/maplibre` folder and select any notebook to run

7.8.2. Exercise 2: Exploring the Environment

1. In the same Docker container from Exercise 1, create a new notebook
2. Import some geospatial libraries to verify they're installed:

```
import geopandas as gpd
import rasterio
import leafmap
print("All libraries imported successfully!")
```

3. Create a simple map using leafmap:

```
import leafmap.maplibregl as leafmap
m = leafmap.Map(projection="globe")
m
```

4. Save this notebook as "leafmap_test.ipynb"

7.8.3. Exercise 3: Working with Different Ports

1. Stop your current Docker container (Ctrl+C)
2. Try to run the container again using port 8889 instead of 8888
3. Access JupyterLab using the new port
4. Verify that your previous notebooks are still available

7.8.4. Exercise 4: Docker Commands Practice

1. Use `docker ps` to see running containers
2. Use `docker images` to see downloaded images
3. Stop your container using `docker stop` (you'll need to find the container name with `docker ps` first)
4. Use `docker ps -a` to see your stopped container
5. Start a new container and note how it gets a different container ID

These exercises will give you hands-on experience with the fundamental Docker operations you'll use in geospatial programming projects.

Section II: Python Programming Fundamentals

Chapter 8. Variables and Data Types

8.1. Introduction

In Python programming, variables and data types are the fundamental building blocks that allow us to store and work with information. Just as a geographer needs to understand basic map elements before creating complex spatial analyses, a programmer must master variables and data types before working with geospatial data.

Variables act as containers that hold information. They allow us to store, access, and modify data throughout our programs. In geospatial work, variables might store coordinates for a specific location, elevation measurements, or place names. Data types define what kind of information we're working with and what operations we can perform on that data.

Understanding variables and data types is especially important in geospatial programming because spatial data comes in many different forms: coordinates are typically decimal numbers, place names are text, and true/false flags might indicate data quality. Learning to work with these different types of data effectively is essential for any geospatial analysis.

8.2. Learning Objectives

By the end of this chapter, you will be able to:

- Create and use variables in Python following proper naming conventions
- Understand and work with Python's main data types: integers, floats, strings, booleans, lists, and dictionaries
- Apply variables and data types to store and manipulate geospatial information such as coordinates, place names, and feature attributes
- Perform basic operations on different data types relevant to geospatial work
- Organize spatial data using lists and dictionaries for simple geospatial applications

8.3. Variables in Python

In Python, a variable is a name that refers to a piece of data stored in the computer's memory. Think of variables as labeled containers that hold information we want to use in our programs.

Let's start by creating a simple variable that represents the number of spatial points in a dataset:

```
num_points = 120
```

This variable `num_points` now holds the integer value 120, which we can use in our calculations or logic.

To view the value of the variable, we can use the `print()` function:

```
print(num_points)
```

Alternatively, we can simply type the variable name in a code cell and run the cell to display the value of the variable:

```
num_points
```

8.3.1. Variable Assignment

Python allows us to easily assign new values to variables and even change the type of data a variable holds:

```
# Start with a number
location_data = 42.3601

# Change to text
location_data = "Boston"

# Change to a list of coordinates
location_data = [42.3601, -71.0589]

print(location_data)
```

8.4. Naming Variables

When naming variables, you should follow these rules:

- Variable names must start with a letter or an underscore, such as `_`.
- The remainder of the variable name can consist of letters, numbers, and underscores.
- Variable names are case-sensitive, so `num_points` and `Num_Points` are different variables.
- Variable names should be descriptive and meaningful, such as `num_points` instead of `n`.
- Avoid using Python keywords and built-in functions as variable names, such as `print`, `sum`, `list`, `dict`, `str`, `int`, `float`, `bool`, `if`, `else`, `for`, `while`, `def`, `class`, etc.

8.4.1. Good Naming Examples for Geospatial Data

The following are good examples of variable names for geospatial data:

```
# Good variable names for geospatial data
latitude = 42.3601
longitude = -71.0589
elevation = 147.2
city_name = "Boston"
population = 685094
coordinate_system = "WGS 84"
```

8.4.2. Poor Naming Examples to Avoid

The following are poor examples of variable names for geospatial data:

```
# Poor variable names - avoid these
x = 42.3601      # Too generic
data = "Boston"  # Too vague
temp = 25.6      # Ambiguous - temperature or temporary?
l = [42.36, -71.06] # Single letter variables are hard to understand
```

8.5. Data Types

Python supports various data types, which are essential to understand before working with geospatial data. The most common data types include:

a) Integers (int): These are whole numbers, e.g., 1, 120, -5

```
num_features = 500 # Represents the number of features in a geospatial dataset
```

b) Floating-point numbers (float): These are numbers with a decimal point, e.g., 3.14, -0.001, 100.0.

```
latitude = 35.6895 # Represents the latitude of a point on Earth's surface
longitude = 139.6917 # Represents the longitude of a point on Earth's surface
```

c) Strings (str): Strings are sequences of characters, e.g., "Hello", "Geospatial Data", "Lat/Long"

```
coordinate_system = "WGS 84" # Represents a commonly used coordinate system
```

Strings can be enclosed in single quotes (`'`) or double quotes (`"`). You can also use triple quotes (`'''` or `"""`) for multiline strings.

d) Booleans (bool): Booleans represent one of two values: True or False

```
is_georeferenced = True # Represents whether a dataset is georeferenced or not
```

e) Lists: Lists are ordered collections of items, which can be of any data type.

```
coordinates = [
    35.6895,
    139.6917,
] # A list representing latitude and longitude of a point
```

f) Dictionaries (dict): Dictionaries are collections of key-value pairs.

```
feature_attributes = {
    "name": "Mount Fuji",
    "height_meters": 3776,
    "type": "Stratovolcano",
```

```
}    "location": [35.3606, 138.7274],
```

8.6. Escape Characters

Escape characters are used to insert characters that are illegal in a string. For example, you can use the escape character `\n` to insert a new line in a string.

```
print("Hello World!\nThis is a Python script.")
```

Another common escape character is `\t`, which inserts a tab in a string.

```
print("This is the first line.\n\tThis is the second line. It is indented.")
```

The output of the above code should look like this:

```
This is the first line.
    This is the second line. It is indented.
```

If you want to include a single quote in a string, you can wrap the string in double quotes.

```
print("What's your name?")
```

Alternatively, you can use the escape character `\'` to include a single quote in a string.

```
print('What\'s your name?')
```

8.7. Comments in Python

Comments are used to explain the code and make it more readable. In Python, comments start with the `#` symbol. Everything after the `#` symbol on a line is ignored by the Python interpreter.

```
# This is a comment
num_points = 120 # This is an inline comment
```

8.8. Working with Variables and Data Types

Now, let's do some basic operations with these variables.

Adding a constant to the number of features:

```
num_features += 20
print("Updated number of features:", num_features)
```

Converting latitude from degrees to radians (required for some geospatial calculations):

```
import math

latitude = 35.6895
latitude_radians = math.radians(latitude)
print("Latitude in radians:", latitude_radians)
```

We first import the `math` module, which contains the `radians()` function. Then we convert the latitude from degrees to radians.

Adding new coordinates to the list of coordinates using the `append()` method:

```
coordinates = [35.6895, 139.6917]
coordinates.append(34.0522) # Adding latitude of Los Angeles
coordinates.append(-118.2437) # Adding longitude of Los Angeles
print("Updated coordinates:", coordinates)
```

Accessing dictionary elements using the `[]` operator:

```
mount_fuji_name = feature_attributes["name"]
mount_fuji_height = feature_attributes["height_meters"]
print(f"{mount_fuji_name} is {mount_fuji_height} meters high.")
```

8.9. Basic String Operations

Strings in Python come with many built-in methods that allow you to manipulate and format text data. This is particularly useful in geospatial work when dealing with place names, addresses, or data labels. Here are some common string operations:

8.9.1. Changing Case

You can convert strings to different cases using built-in methods:

```
city_name = "San Francisco"

# Convert to lowercase
city_lowercase = city_name.lower()
print("Lowercase:", city_lowercase)

# Convert to uppercase
city_uppercase = city_name.upper()
print("Uppercase:", city_uppercase)

# Convert to title case (first letter of each word capitalized)
city_title = city_name.title()
print("Title case:", city_title)
```

8.9.2. Replacing Text

The `replace()` method allows you to substitute parts of a string with new text:

```
original_city = "San Francisco"
new_city = original_city.replace("San", "Los")
print("Original:", original_city)
print("Modified:", new_city)
```

8.9.3. Other Useful String Methods

Here are a few more string methods that are commonly used in geospatial programming:

```
location_data = "  Mount Everest  "

# Remove whitespace from beginning and end
clean_location = location_data.strip()
print("Cleaned:", clean_location)
```

We will cover more string operations in the [String Operations](#) chapter.

8.10. Key Takeaways

Understanding Python variables and data types is crucial in geospatial programming. As you proceed with more complex analyses, these concepts will serve as the foundation for your work. Continue practicing by experimenting with different data types and operations in a geospatial context.

Happy coding!

8.11. Exercises

8.11.1. Exercise 1: Variable Assignment and Basic Operations

Create variables to store the following geospatial data:

- The latitude and longitude of New York City: 40.7128, -74.0060.
- The population of New York City: 8,336,817.
- The area of New York City in square kilometers: 783.8.

Perform the following tasks:

1. Calculate and print the population density of New York City (population per square kilometer).
2. Print the coordinates in the format "Latitude: [latitude], Longitude: [longitude]".

8.11.2. Exercise 2: Working with Strings

Create a string variable to store the name of a city, such as “New York City”. Perform the following operations:

1. Convert the string to lowercase and print the result.
2. Convert the string to uppercase and print the result.
3. Replace “New York” with “Los Angeles” in the city name and print the new string.



Chapter 9. Python Data Structures

9.1. Introduction

Data structures are the building blocks that allow us to organize and store multiple pieces of related information together. While individual variables hold single values like a coordinate or place name, data structures let us group multiple values in meaningful ways. This becomes essential in geospatial programming where we often work with collections of coordinates, lists of place names, sets of unique identifiers, and structured attribute information.

Python provides several built-in data structures that are particularly useful for geospatial work: tuples for storing fixed coordinate pairs, lists for sequences of locations along a path, sets for collections of unique identifiers, and dictionaries for organizing feature attributes. Understanding when and how to use each of these structures is fundamental to effective geospatial programming.

These data structures serve as the foundation for more complex geospatial operations. Whether you're tracking a GPS route with a list of coordinates, storing unique country codes in a set, or organizing feature attributes in a dictionary, mastering these basic structures will enable you to work efficiently with spatial data.

9.2. Learning Objectives

By the end of this chapter, you should be able to:

- Understand the characteristics and use cases of Python tuples, lists, sets, and dictionaries
- Apply these data structures to store and manipulate geospatial data, such as coordinates, paths, and attribute information
- Differentiate between mutable and immutable data structures and choose the appropriate structure for different geospatial tasks
- Perform common operations on these data structures, including indexing, slicing, adding/removing elements, and updating values
- Utilize dictionaries to manage geospatial feature attributes and understand the importance of key-value pairs in geospatial data management

9.3. Tuples

Tuples are immutable sequences, which means that once a tuple is created, you cannot change, add, or remove its elements. This immutability makes tuples perfect for storing data that should remain constant throughout your program's execution. In geospatial programming, tuples are commonly used to represent coordinate pairs, since a point's latitude and longitude typically shouldn't change once defined.

The immutable nature of tuples provides several advantages: they're memory-efficient, can be used as dictionary keys (which we'll see later), and help prevent accidental modification of important spatial reference data. When you want to represent a fixed geographic location, a tuple is often the best choice.

9.3.1. Creating and Using Tuples

Tuples are created using parentheses `()` with elements separated by commas. Let's create a tuple to represent the coordinates of Tokyo:


```
tokyo_point = (
    35.6895,
    139.6917,
) # Tuple representing Tokyo's coordinates (latitude, longitude)
print(f"Tokyo coordinates: {tokyo_point}")
```

9.3.2. Accessing Tuple Elements

You can access individual elements in a tuple using indexing, where the first element is at index 0:

```
latitude = tokyo_point[0]
longitude = tokyo_point[1]
print(f"Latitude: {latitude}")
print(f"Longitude: {longitude}")
```

9.3.3. Tuple Unpacking

Python provides a convenient way to extract tuple elements into separate variables called “unpacking”:

```
lat, lon = tokyo_point # Unpacking the tuple into two variables
print(f"Tokyo is located at {lat}°N, {lon}°E")
```

9.3.4. Multiple Coordinate Points

You can create tuples to represent different types of geographic features:

```
# Different geographic locations as tuples
new_york = (40.7128, -74.0060)
london = (51.5074, -0.1278)
sydney = (-33.8688, 151.2093)

print(f"New York: {new_york}")
print(f"London: {london}")
print(f"Sydney: {sydney}")
```

9.4. Lists

Lists are ordered, mutable sequences that can store multiple items in a single container. Unlike tuples, lists are mutable, which means you can change, add, or remove elements after the list has been created. This flexibility makes lists incredibly useful for geospatial applications where you need to build collections of data dynamically, such as tracking a GPS route, storing elevation measurements along a transect, or maintaining a collection of waypoints.

Lists maintain the order of elements, which is crucial for geospatial applications where sequence matters. For example, when representing a path or route, the order of waypoints determines the direction of travel.

Lists can store different types of data (numbers, strings, tuples, or even other lists), making them versatile containers for complex geospatial information.

9.4.1. Creating Lists

Lists are created using square brackets `[]` with elements separated by commas. Here are some examples relevant to geospatial work:

```
# A list of coordinate tuples representing a travel route
route = [
    (35.6895, 139.6917), # Tokyo
    (34.0522, -118.2437), # Los Angeles
    (51.5074, -0.1278), # London
]
print("Travel route:", route)
```

```
# A list of elevation measurements (in meters)
elevations = [120.5, 135.2, 150.8, 168.3, 185.7, 203.1]
print("Elevation profile:", elevations)
```

```
# A list of city names
cities = ["Tokyo", "Los Angeles", "London", "Paris"]
print("Cities to visit:", cities)
```

9.4.2. Adding Elements to Lists

One of the key advantages of lists is the ability to add new elements dynamically using the `append()` method. This is particularly useful when you need to build a list of coordinates or other data points as you process them.

```
# Add Paris to our travel route
route.append((48.8566, 2.3522)) # Adding Paris coordinates
print("Updated route:", route)
```

```
# Add a new elevation measurement
elevations.append(221.4)
print("Updated elevations:", elevations)
```

Note that the `append()` method modifies the list in place, meaning it directly changes the original list rather than returning a new list.

9.4.3. Accessing List Elements

You can access individual elements using indexing (starting from 0) or retrieve multiple elements using slicing:

```
# Access the first city in our route
first_stop = route[0]
print(f"First stop: {first_stop}")

# Access the last city using negative indexing
last_stop = route[-1]
print(f"Last stop: {last_stop}")
```

9.4.4. List Slicing

Slicing allows you to extract portions of a list, which is useful for analyzing segments of routes or data:

```
# Get the first two stops of our route
first_two_stops = route[:2]
print("First two stops:", first_two_stops)

# Get the middle portion of elevation data
middle_elevations = elevations[2:5]
print("Middle elevation readings:", middle_elevations)
```

9.4.5. Useful List Operations

Lists provide many helpful methods for working with geospatial data, such as finding the number of waypoints in a route, calculating the highest elevation, or computing the average elevation. These operations are essential for analyzing and visualizing geospatial data.

```
# Find the number of waypoints in our route
num_waypoints = len(route)
print(f"Number of waypoints: {num_waypoints}")

# Find the highest elevation
max_elevation = max(elevations)
print(f"Highest elevation: {max_elevation} meters")

# Calculate average elevation
avg_elevation = sum(elevations) / len(elevations)
print(f"Average elevation: {avg_elevation:.1f} meters")
```

9.5. Sets

Sets are unordered collections of unique elements, meaning they automatically eliminate duplicates and don't maintain any particular order. This makes sets incredibly useful in geospatial programming when you need to work with unique identifiers, remove duplicate entries from datasets, or perform operations like finding common elements between different spatial datasets.

Sets are particularly valuable when working with categorical spatial data. For example, you might want to track unique country codes in a global dataset, identify distinct land cover types in a study area, or maintain a collection of unique coordinate system identifiers. The automatic duplicate removal feature of sets saves you from having to manually check for and remove repeated values.

9.5.1. Creating Sets

You can create sets in several ways. You can use curly braces `{}` to create a set, or use the `set()` function to convert a list or other iterable into a set. Here are examples relevant to geospatial work:

```
# Create a set of geographic regions
regions_visited = {"North America", "Europe", "Asia"}
print("Regions visited:", regions_visited)
```

```
# Create a set from a list (automatically removes duplicates)
country_codes = ["US", "CA", "MX", "US", "CA"] # Notice the duplicates
unique_codes = set(country_codes)
print("Original list:", country_codes)
print("Unique country codes:", unique_codes)
```

The output should look like this:

```
Original list: ['US', 'CA', 'MX', 'US', 'CA']
Unique country codes: {'MX', 'CA', 'US'}
```

```
# Create a set of coordinate system codes
crs_codes = {"EPSG:4326", "EPSG:3857", "EPSG:32633"}
print("Coordinate reference systems:", crs_codes)
```

9.5.2. Adding Elements to Sets

You can add new elements to a set, but duplicates will be automatically ignored:

```
# Add a new region
print("Original set:", regions_visited)
regions_visited.add("Africa")
print("After adding Africa:", regions_visited)

# Try to add a duplicate region
regions_visited.add("Europe") # This won't change the set
print("After trying to add Europe again:", regions_visited)
```

The output should look like this:

```
Original set: {'Asia', 'Europe', 'North America'}
After adding Africa: {'Asia', 'Europe', 'North America', 'Africa'}
After trying to add Europe again: {'Asia', 'Europe', 'North America', 'Africa'}
```

9.5.3. Practical Set Operations

Sets provide useful operations for comparing different spatial datasets. For example, you might want to find the common countries between two different datasets, or identify the unique countries in a dataset. The `intersection()` method returns a set of elements that are common to both sets, while the `union()` method returns a set of all elements from both sets.

```
# Two different survey areas with their observed species
area_a_species = {"oak", "pine", "maple", "birch"}
area_b_species = {"pine", "birch", "cedar", "fir"}

print("Species in Area A:", area_a_species)
print("Species in Area B:", area_b_species)

# Find species common to both areas
common_species = area_a_species.intersection(area_b_species)
print("Species found in both areas:", common_species)

# Find species unique to Area A
unique_to_a = area_a_species - area_b_species
print("Species only in Area A:", unique_to_a)

# Find all species across both areas
all_species = area_a_species.union(area_b_species)
print("All species found:", all_species)
```

9.5.4. Checking Set Membership

Sets provide very fast membership testing, which is useful for checking if a particular value exists in your collection:

```
# Check if we've visited a particular region
if "Asia" in regions_visited:
    print("We have visited Asia")

if "Antarctica" in regions_visited:
    print("We have visited Antarctica")
else:
    print("Antarctica not yet visited")
```

9.6. Dictionaries

Dictionaries are collections of key-value pairs where each key is unique and maps to a specific value. This structure is perfect for storing related information about geographic features, where you need to associate descriptive attributes with specific identifiers. In geospatial programming, dictionaries are extensively used to store feature attributes, metadata about datasets, configuration settings, and any situation where you need to organize information by meaningful names rather than numeric positions.

Think of dictionaries as lookup tables or filing systems where you can quickly find information using a descriptive key. For example, instead of remembering that population data is stored in position 2 of a list, you can simply use the key “population” to access that information directly. This makes your code more readable and less prone to errors.

9.6.1. Creating Dictionaries

Dictionaries are created using curly braces `{}` with key-value pairs separated by commas and keys separated from values by colons:

```
# Dictionary storing attributes of a city
tokyo_info = {
    "name": "Tokyo",
    "population": 13929286,
    "coordinates": (35.6895, 139.6917),
    "country": "Japan",
    "established": 1603,
}
print("Tokyo information:", tokyo_info)
```

```
# Dictionary for a geographic survey point
survey_point = {
    "point_id": "SP001",
    "latitude": 40.7589,
    "longitude": -73.9851,
    "elevation": 87.3,
    "land_cover": "urban",
    "date_surveyed": "2023-06-15",
}
print("Survey point data:", survey_point)
```

9.6.2. Accessing Dictionary Values

You can access values using their keys in square brackets:

```
# Access specific information about Tokyo
city_name = tokyo_info["name"]
city_population = tokyo_info["population"]
city_coords = tokyo_info["coordinates"]
```

```
print(f"City: {city_name}")
print(f"Population: {city_population:,}")
print(f"Coordinates: {city_coords}")
```

Keep in mind that if you try to access a key that doesn't exist in a dictionary, you'll get a `KeyError` error. To avoid this, you can use the `get()` method as introduced below to safely access a key and provide a default value if the key doesn't exist.

9.6.3. Safe Access with `get()`

The `get()` method provides a safe way to access dictionary values, allowing you to specify a default value if the key doesn't exist:

```
# Safe access to dictionary values
area = tokyo_info.get("area_km2", "Not specified")
timezone = tokyo_info.get("timezone", "Unknown")

print(f"Area: {area}")
print(f"Timezone: {timezone}")
```

9.6.4. Adding and Updating Values

You can add new key-value pairs or update existing ones:

```
# Add new information to our Tokyo dictionary
tokyo_info["area_km2"] = 2191
tokyo_info["timezone"] = "JST"
tokyo_info["population"] = 14000000 # Update population

print("Updated Tokyo info:", tokyo_info)
```

9.6.5. Working with Geographic Feature Collections

Dictionaries are excellent for organizing multiple geographic features:

```
# Collection of world capitals
world_capitals = {
    "Japan": {
        "capital": "Tokyo",
        "coordinates": (35.6895, 139.6917),
        "population": 13929286,
    },
    "France": {
        "capital": "Paris",
```

```

        "coordinates": (48.8566, 2.3522),
        "population": 2161000,
    },
    "UK": {
        "capital": "London",
        "coordinates": (51.5074, -0.1278),
        "population": 8982000,
    },
}

# Access information about France's capital
france_info = world_capitals["France"]
print(f"France's capital: {france_info['capital']}")
print(f"Population: {france_info['population']:,}")

```

As you can see from the example above, a dictionary can be nested within another dictionary. This is useful when you need to store information about a city, and then store information about the city's population, coordinates, and country.

9.6.6. Dictionary Methods for Data Exploration

Dictionaries provide useful methods for exploring your data, such as `keys()`, `values()`, and `items()`. These methods return views of the dictionary's keys, values, and key-value pairs, respectively. They are useful for iterating over the dictionary and for checking if a key exists in the dictionary. We will learn more about these methods in the **Loops and Conditional Statements** chapter. Below is an example of how to use these methods.

```

# Explore the structure of our Tokyo dictionary
print("Keys in tokyo_info:", list(tokyo_info.keys()))
print("Values in tokyo_info:", list(tokyo_info.values()))

# Check if a key exists
if "coordinates" in tokyo_info:
    print("Coordinate information is available")

# Get all countries in our capitals dictionary
countries = list(world_capitals.keys())
print("Countries in our database:", countries)

```

9.6.7. Practical Example: GPS Waypoint Management

Here's a practical example of using dictionaries to manage GPS waypoints:

```

# GPS waypoint with comprehensive metadata
waypoint = {
    "id": "WP001",

```



```

    "name": "Trail Start",
    "latitude": 45.3311,
    "longitude": -121.7113,
    "elevation": 1200,
    "description": "Beginning of Pacific Crest Trail section",
    "waypoint_type": "trailhead",
    "facilities": ["parking", "restrooms", "water"],
    "difficulty": "easy",
}

print(f"Waypoint: {waypoint['name']}")
print(f"Location: {waypoint['latitude']}, {waypoint['longitude']}")
print(f"Elevation: {waypoint['elevation']} meters")
print(f"Available facilities: {' '.join(waypoint['facilities'])}")

```

9.7. Data Structure Selection Guide

Choosing the right data structure for your geospatial tasks is crucial for writing efficient and maintainable code. Here's a quick guide to help you decide:

9.7.1. When to Use Each Data Structure

Use Tuples when:

- You have a fixed pair or group of values that shouldn't change (like coordinates)
- You need to store data that will be used as dictionary keys
- You want to ensure data integrity by preventing accidental modifications

Use Lists when:

- You need an ordered collection that can change (like a GPS route or waypoint sequence)
- You want to add, remove, or modify elements dynamically
- Order matters for your application (like steps in a workflow)

Use Sets when:

- You need to eliminate duplicates from your data
- You want to perform operations like finding common elements between datasets
- Order doesn't matter, but uniqueness does

Use Dictionaries when:

- You need to associate descriptive names with values (like feature attributes)
- You want fast lookup of information by key
- You're organizing complex, structured data about geographic features

9.8. Key Takeaways

Python's data structures form the foundation of effective geospatial programming. By understanding when and how to use tuples, lists, sets, and dictionaries, you can organize spatial data efficiently and write more readable code.

- **Tuples** provide immutable containers perfect for coordinate pairs and fixed reference data
- **Lists** offer flexible, ordered collections ideal for sequences like routes and measurement series
- **Sets** automatically handle uniqueness and provide powerful operations for comparing datasets
- **Dictionaries** organize complex attribute information with meaningful, descriptive keys

As you work with real geospatial data, you'll often combine these structures - for example, using a list of dictionaries to represent multiple geographic features, or storing coordinate tuples within a dictionary's values. Practice with these fundamental structures will prepare you for more advanced geospatial programming concepts and libraries.

The exercises below will help reinforce these concepts and give you hands-on experience applying these data structures to common geospatial scenarios.

9.9. Exercises

9.9.1. Exercise 1: Using Lists

Create a list of tuples, where each tuple contains the name of a city and its corresponding latitude and longitude:

- New York City: (40.7128, -74.0060)
- Los Angeles: (34.0522, -118.2437)
- Chicago: (41.8781, -87.6298)

Perform the following tasks:

1. Add a new city (e.g., Miami: (25.7617, -80.1918)) to the list.
2. Print the entire list of cities.
3. Slice the list to print only the first two cities.

9.9.2. Exercise 2: Using Tuples

Create a tuple to store the coordinates (latitude, longitude) of the Eiffel Tower: (48.8584, 2.2945). Perform the following tasks:

1. Access and print the latitude and longitude values from the tuple.
2. Try to change the latitude value to 48.8585. What happens? Explain why.

9.9.3. Exercise 3: Working with Sets

Create a set of countries you have visited, such as {"USA", "France", "Germany"}. Perform the following tasks:

1. Add a new country to the set.
2. Try to add the same country again. What happens?
3. Print the updated set.

9.9.4. Exercise 4: Working with Dictionaries

Create a dictionary to store information about a specific geospatial feature, such as a river:

- Name: “Amazon River”
- Length: 6400 km
- Countries: [“Brazil”, “Peru”, “Colombia”]

Perform the following tasks:

1. Add a new key-value pair to the dictionary to store the river’s average discharge (e.g., 209,000 m³/s).
2. Update the length of the river to 6992 km.
3. Print the dictionary.

9.9.5. Exercise 5: Nested Data Structures

Create a dictionary to represent a city that contains the city’s name, population, and coordinates (latitude, longitude):

- Name: “Tokyo”
- Population: 13,515,271
- Coordinates: (35.6895, 139.6917)

Perform the following tasks:

1. Access and print the population of the city.
2. Access and print the city’s latitude.
3. Update the population to 14,000,000 and print the updated dictionary.

Chapter 10. String Operations

10.1. Introduction

Strings are sequences of characters that represent textual information in Python programs. In geospatial programming, strings play a crucial role in handling various types of text-based data: place names, coordinate system definitions, file paths, data extracted from text files, and even coordinate values that come in text format from GPS devices or web services.

Working effectively with strings is essential for many common geospatial tasks. You might need to clean messy place names from a dataset, parse coordinates from a GPS log file, format location information for display on a map, or build file paths to access geographic data files. Understanding how to manipulate strings gives you the tools to handle these text-processing challenges efficiently.

Python provides a rich set of built-in string operations and methods that make working with textual data straightforward and powerful. Whether you're concatenating location names, extracting numeric values from coordinate strings, or formatting output for reports, mastering these fundamental string operations will significantly enhance your ability to work with geospatial data.

10.2. Learning Objectives

By the end of this chapter, you should be able to:

- Create and manipulate strings in Python, including concatenation and repetition
- Apply string methods such as `lower()`, `upper()`, `strip()`, `replace()`, and `split()` to process geospatial data
- Format strings using the `format()` method and f-strings to include variable data within strings
- Parse and extract specific information from strings, such as coordinates or location names
- Utilize string operations in practical geospatial tasks, enhancing your ability to work with and manage geographic data

10.3. Creating and Manipulating Strings

Strings are fundamental data types in Python that store sequences of characters. Understanding how to create and manipulate strings is essential for handling textual geospatial data effectively.

10.3.1. Creating Strings

Strings can be created by enclosing characters in single quotes (`'`), double quotes (`"`), or triple quotes (`'''` or `"""`) for multi-line strings:

```
# Different ways to create strings
location_name = "Mount Everest" # Using double quotes
country = 'Nepal' # Using single quotes
description = """Mount Everest is the highest peak
in the world, located in the Himalayas.""" # Multi-line string

print(f"Location: {location_name}")
```

```
print(f"Country: {country}")
print(f"Description: {description}")
```

10.3.2. String Concatenation

Concatenation means joining strings together. This is useful for building location descriptions, file paths, or combining separate pieces of geographic information. Concatenation is done using the `+` operator.

```
# Basic concatenation using the + operator
location_full = location_name + ", " + country
print(f"Full location: {location_full}")

# Building a file path
data_folder = "geographic_data"
filename = "mountain_peaks.csv"
file_path = data_folder + "/" + filename
print(f"File path: {file_path}")
```

10.3.3. String Repetition

The `*` operator allows you to repeat strings, which can be useful for creating separators, borders, or formatting output:

```
# Create visual separators
separator = "-" * 30
print(separator)
print("Geographic Data Report")
print(separator)

# Create formatted spacing
tab_space = " " * 4
print(f"Location:{tab_space}{location_name}")
print(f"Elevation:{tab_space}8,848 meters")
```

10.3.4. String Length and Basic Properties

You can get information about strings using built-in functions. For example, you can get the length of a string using the `len()` function. You can also check if a string contains only letters using the `isalpha()` method. To check if a string contains only digits, you can use the `isdigit()` method. Below is an example of how to use these functions:

```
# Get the length of a string
location_length = len(location_name)
print(f"The location name '{location_name}' has {location_length} characters")
```

```
# Check if a string contains only letters
city_name = "SanFrancisco"
print(f"Is '{city_name}' alphabetic? {city_name.isalpha()}")

# Check if a string contains only digits (useful for coordinate validation)
zip_code = "94102"
print(f"Is '{zip_code}' numeric? {zip_code.isdigit()}")
```

10.3.5. Building Dynamic Content

Strings are often built dynamically based on variable data, which is common when working with geographic datasets. For example, you might want to join a list of city names with a comma and a space between them. This can be done using the `join()` method with a string as the separator.

```
# Building dynamic location descriptions
latitude = 27.9881
longitude = 86.9250
elevation = 8848

location_info = (
    location_name
    + " is located at coordinates "
    + str(latitude)
    + ", "
    + str(longitude)
)
print(location_info)

# A more complex example - building a geographic summary
cities = ["Kathmandu", "Pokhara", "Lalitpur"]
summary = "Major cities in " + country + " include: " + ", ".join(cities)
print(summary)
```

10.4. String Methods for Geospatial Data

Python provides powerful built-in methods that allow you to transform and manipulate strings. These methods are particularly useful in geospatial programming for cleaning data, standardizing formats, and extracting information from text-based geographic data.

10.4.1. Case Conversion Methods

Converting between uppercase and lowercase is essential for standardizing geographic data, especially when working with place names from different sources. The `upper()`, `lower()`, `title()`, and `capitalize()` methods are particularly useful for this purpose.

```
# Case conversion examples
mountain_name = "Mount Everest"

# Convert to different cases
print(f"Original: {mountain_name}")
print(f"Uppercase: {mountain_name.upper()}")
print(f"Lowercase: {mountain_name.lower()}")
print(f"Title case: {mountain_name.title()}")
print(f"Capitalize: {mountain_name.capitalize()}")
```

The output should look like this:

```
Original: Mount Everest
Uppercase: MOUNT EVEREST
Lowercase: mount everest
Title case: Mount Everest
Capitalize: Mount everest
```

10.4.2. Whitespace Removal Methods

Geographic data often comes with unwanted spaces that need to be cleaned. The strip methods help remove these spaces. There are three methods: `strip()`, `lstrip()`, and `rstrip()`. The `strip()` method removes both leading and trailing spaces, while the `lstrip()` method removes leading spaces and the `rstrip()` method removes trailing spaces. Here is an example:

```
# Whitespace removal examples
messy_location = "  San Francisco  "
messy_left = "  Los Angeles"
messy_right = "Chicago  "

print(f"Original: '{messy_location}'")
print(f"strip(): '{messy_location.strip()}'") # Remove both sides
print(f"lstrip(): '{messy_left.lstrip()}'") # Remove left side
print(f"rstrip(): '{messy_right.rstrip()}'") # Remove right side
```

10.4.3. String Replacement

The `replace()` method allows you to substitute parts of strings, which is useful for correcting data or updating information:

```
# Basic replacement
location = "Mount Everest, Nepal"
updated_location = location.replace("Everest", "Kilimanjaro")
print(f"Original: {location}")
print(f"Updated: {updated_location}")
```

```
# Replace multiple occurrences
path_string = "data/raw_data/geographic_data/raw_data/points.csv"
clean_path = path_string.replace("raw_data/", "")
print(f"Original path: {path_string}")
print(f"Clean path: {clean_path}")
```

10.4.4. String Splitting

The `split()` method breaks strings into lists, which is extremely useful for parsing structured geographic data. For example, you might have a string that contains a list of city names separated by commas, and you want to split it into a list of city names. The `split()` method takes a separator as an argument, and returns a list of strings. The default separator is a space, but you can specify a different separator if needed. The example below shows how to split a string into a list of city names using the `split()` method with a comma as the separator:

```
# Basic splitting
location_full = "Mount Everest, Nepal, Asia"
location_parts = location_full.split(", ")
print(f"Original: {location_full}")
print(f"Split into parts: {location_parts}")

# Extract individual components
mountain, country, continent = location_parts
print(f"Mountain: {mountain}")
print(f"Country: {country}")
print(f"Continent: {continent}")
```

The example below shows how to split a string into a list of coordinates:

```
# Splitting coordinate strings
coordinate_string = "40.7128,-74.0060"
lat_str, lon_str = coordinate_string.split(",")
latitude = float(lat_str)
longitude = float(lon_str)
print(f"Parsed coordinates: Lat={latitude}, Lon={longitude}")
```

The example below shows how to split a file path into a list of path components using the `split()` method with a forward slash as the separator:

```
# Splitting file paths
file_path = "data/geographic/cities/world_cities.csv"
path_components = file_path.split("/")
print(f"Path components: {path_components}")
print(f"Filename: {path_components[-1]}") # Last component is the filename
```


10.4.5. String Joining

The `join()` method combines lists of strings into single strings, which is the reverse of splitting. For example, you might have a list of city names, and you want to join them into a single string with a comma and a space between each city name. To use the `join()` method, you need to provide a string as the separator. The example below shows how to join a list of city names into a single string with a comma and a space between each city name:

```
# Basic joining
city_names = ["San Francisco", "New York", "Tokyo"]
city_name = ", ".join(city_names)
print(f"Joined city name: {city_name}")
```

The example below shows how to join a list of file paths into a single file path using the `join()` method with a forward slash as the separator:

```
# Creating file paths
path_parts = ["data", "geographic", "elevation", "dem.tif"]
full_path = "/".join(path_parts)
print(f"Full path: {full_path}")
```

The example below shows how to join a list of coordinates into a string:

```
# Practical example: creating coordinate strings
coordinates = ["40.7128", "-74.0060"]
coordinate_string = ", ".join(coordinates)
print(f"Coordinate string: {coordinate_string}")
```

10.5. String Formatting

String formatting allows you to create strings that incorporate variable values, which is essential for generating reports, creating file names, building queries, and displaying geographic information. Python provides several powerful formatting methods that make it easy to create well-structured text output.

10.5.1. F-String Formatting (Recommended)

F-strings (formatted string literals) are the most modern and readable way to format strings in Python. They're particularly useful for creating geographic reports and displaying coordinate information. To use f-strings, you can use the `f` prefix before the string, and then use curly braces to insert the variable values. The example below shows how to use f-strings to format a string with a variable value:

```
# Basic f-string formatting with geographic data
location = "Mount Everest"
latitude = 27.9881
longitude = 86.9250
elevation = 8848
```

```
# Simple variable insertion
location_info = f"Location: {location}"
print(location_info)

# Multiple variables
coordinates = f"Coordinates: ({latitude}, {longitude})"
print(coordinates)

# Complete geographic summary
summary = f"{location} is located at {latitude}°N, {longitude}°E with an
elevation of {elevation} meters"
print(summary)
```

10.5.2. Formatting Numbers in Strings

When working with geographic data, you often need to control how numbers are displayed. For example, you might want to display a coordinate with a certain number of decimal places, or a number with a certain number of significant figures. To do this, you can use f-strings to format numbers. The format specifier is a colon `:` followed by the number of decimal places or significant figures. The example below shows how to format a number with a certain number of decimal places:

```
# Controlling decimal places for coordinates
precise_lat = 40.712776
precise_lon = -74.005974

# Round to different decimal places
coords_2_places = f"Coordinates: ({precise_lat:.2f}, {precise_lon:.2f})"
coords_4_places = f"Coordinates: ({precise_lat:.4f}, {precise_lon:.4f})"

print(coords_2_places)
print(coords_4_places)

# Adding thousands separators for large numbers
population = 8336817
area_sqkm = 783.8

formatted_stats = f"NYC Population: {population:,} people, Area: {area_sqkm:.1f}
km²"
print(formatted_stats)
```

10.5.3. Legacy Formatting Methods

While f-strings are preferred, you should be familiar with older formatting methods that you might encounter in older code. The example below shows how to format a number with a certain number of decimal places using the `format()` method. Instead of using curly braces, you use curly braces with a

number inside to indicate the position of the variable to be formatted. The example below shows how to format a number with a certain number of decimal places:

```
# Using .format() method
location = "San Francisco"
lat = 37.7749
lon = -122.4194

# Basic format method
formatted_1 = "Location: {} at coordinates ({}, {})".format(location, lat, lon)
print(formatted_1)

# With positional arguments
formatted_2 = "Location: {0} at coordinates ({1}, {2})".format(location, lat, lon)
print(formatted_2)

# With named arguments
formatted_3 = "Location: {name} at coordinates ({latitude}, {longitude})".format(
    name=location, latitude=lat, longitude=lon
)
print(formatted_3)
```

10.5.4. Practical Formatting Examples

Here are some common geospatial formatting scenarios. For example, you might want to save a coordinate as a string with a certain number of decimal places in a file with timestamp in the filename. The example below shows how to do this:

```
# Creating file names with timestamps and coordinates
import datetime

current_time = datetime.datetime.now()
survey_lat = 45.3311
survey_lon = -121.7113

filename = f"survey_{current_time.strftime('%Y%m%d')}_{{survey_lat:.4f}}N_{{abs(survey_lon):.4f}}W.csv"
print(f"Generated filename: {filename}")

# Creating Well-Known Text (WKT) representations
wkt_point = f"POINT({survey_lon} {survey_lat})"
print(f"WKT Point: {wkt_point}")
```

The example below shows how to build a SQL query with formatting:

```
# Building SQL queries with formatting
table_name = "cities"
min_population = 1000000
region = "North America"

sql_query = f"""SELECT name, latitude, longitude
FROM {table_name}
WHERE population > {min_population:,}
AND region = '{region}'"""

print("Generated SQL Query:")
print(sql_query)
```

10.6. String Operation Decision Guide

Choosing the right string operation for your geospatial task is important for writing efficient and maintainable code:

10.6.1. When to Use Each Operation

Use concatenation (`+` or `join()`) when:

- Building file paths or URLs
- Combining location names with additional information
- Creating simple text outputs

Use string methods (`strip()`, `replace()`, `split()`) when:

- Cleaning messy geographic data
- Standardizing place names or coordinate formats
- Breaking apart structured text data

Use formatting (f-strings) when:

- Creating reports or displaying geographic information
- Building dynamic queries or file names
- Controlling number precision in coordinate displays

10.7. Key Takeaways

String operations form a critical foundation for geospatial data processing. Understanding these fundamental concepts enables you to:

- **Clean and standardize** geographic data from various sources
- **Parse and extract** specific information from text-based spatial data
- **Format and present** geographic information in readable, professional formats
- **Build dynamic content** like file names, queries, and reports

These skills are essential whether you're working with GPS coordinates from field devices, cleaning place names in datasets, formatting output for maps and reports, or processing data from text files. The

combination of basic string operations, formatting techniques, and parsing strategies provides a powerful toolkit for handling the textual aspects of geospatial programming.

As you advance in geospatial programming, you'll find these string manipulation skills valuable when working with coordinate reference systems, parsing spatial data formats, and integrating with web services that return geographic information in text format.

The exercises below will help you practice these concepts with realistic geospatial scenarios.

10.8. Exercises

10.8.1. Exercise 1: Manipulating Geographic Location Strings

- Create a string that represents the name of a geographic feature (e.g., "Amazon River").
- Convert the string to lowercase and then to uppercase.
- Concatenate the string with the name of the country (e.g., "Brazil") to create a full location name.
- Repeat the string three times, separating each repetition with a dash (-).

10.8.2. Exercise 2: Extracting and Formatting Coordinates

- Given a string with the format "latitude, longitude" (e.g., "40.7128N, 74.0060W"), extract the numeric values of latitude and longitude.
- Convert these values to floats and remove the directional indicators (N , S , E , W).
- Format the coordinates into a POINT WKT string (e.g., "POINT(-74.0060 40.7128)").

10.8.3. Exercise 3: Building Dynamic SQL Queries

- Given a table name and a condition, dynamically build an SQL query string.
- Example: If `table_name = "cities"` and `condition = "population > 1000000"`, the query should be `"SELECT * FROM cities WHERE population > 1000000;"`.
- Add additional conditions dynamically, like `AND` clauses.

10.8.4. Exercise 4: String Normalization and Cleaning

- Given a list of city names with inconsistent formatting (e.g., [" new york ", "Los ANGELES", " CHICAGO"]), normalize the names by:
 - Stripping any leading or trailing whitespace.
 - Converting them to title case (e.g., "New York", "Los Angeles", "Chicago").
- Ensure that the output is a clean list of city names.

10.8.5. Exercise 5: Parsing and Extracting Address Information

- Given a string in the format "Street, City, Country" (e.g., "123 Main St, Springfield, USA"), write a function that parses the string into a dictionary with keys street, city, and country.
- The function should return a dictionary like {"street": "123 Main St", "city": "Springfield", "country": "USA"}.

Chapter 11. Loops and Conditional Statements

11.1. Introduction

Looping and control statements are fundamental programming concepts that allow you to automate repetitive tasks and make decisions based on data conditions. In geospatial programming, these concepts become particularly powerful when working with collections of spatial data, such as processing multiple coordinate points, analyzing sets of geographic features, or performing calculations across entire datasets.

Think of loops as a way to avoid writing the same code multiple times. Instead of manually processing each coordinate in a list of 100 GPS points, you can write a loop that automatically handles all of them with just a few lines of code. Control statements, on the other hand, allow your programs to make intelligent decisions. For example, categorizing coordinates by hemisphere, filtering data based on elevation thresholds, or handling different types of geographic features appropriately.

These concepts are essential building blocks that enable you to process real-world geospatial datasets efficiently. Whether you're analyzing GPS tracks, processing survey data, or working with geographic features from a database, loops and control statements will help you automate tedious tasks and build more intelligent, responsive programs.

11.2. Learning Objectives

By the end of this chapter, you should be able to:

- Understand and implement `for` loops to iterate over sequences such as lists and tuples
- Use `while` loops to perform tasks until a specific condition is met
- Apply control statements (`if`, `elif`, `else`) to execute different blocks of code based on data conditions
- Combine loops and control statements to filter, process, and analyze geospatial data
- Develop the ability to automate repetitive geospatial tasks, making your data processing workflows more efficient

11.3. For Loops

For loops are one of the most commonly used programming constructs in geospatial analysis. They allow you to iterate through sequences (e.g., lists, tuples, or strings) and execute the same block of code for each item in the sequence. This eliminates the need to write repetitive code and makes your programs much more efficient and maintainable.

In geospatial programming, for loops are essential for processing collections of spatial data. You might use them to analyze multiple coordinate points, process features from a survey, calculate statistics across multiple measurements, or transform geographic coordinates from one format to another.

11.3.1. Basic For Loop Syntax

The basic structure of a for loop involves specifying a variable to represent each item in the sequence, followed by the sequence you want to iterate through:

```

coordinates = [
    (35.6895, 139.6917),
    (34.0522, -118.2437),
    (51.5074, -0.1278),
] # List of tuples representing coordinates

for lat, lon in coordinates:
    print(f"Latitude: {lat}, Longitude: {lon}")

```

Notice how we use **tuple unpacking** here—each coordinate tuple `(lat, lon)` is automatically unpacked into separate `lat` and `lon` variables. This is a very common and useful pattern when working with coordinate data.

11.3.2. Processing Multiple Data Points

For loops become particularly powerful when you need to perform calculations or transformations on multiple data points. Here's how you can use a loop to compute distances from a reference point:

```

def calculate_distance(lat1, lon1, lat2, lon2):
    # Placeholder for distance calculation logic
    return ((lat2 - lat1) ** 2 + (lon2 - lon1) ** 2) ** 0.5

reference_point = (0, 0) # Reference point (latitude, longitude)

for lat, lon in coordinates:
    distance = calculate_distance(reference_point[0], reference_point[1], lat,
    lon)
    print(f"Distance from {reference_point} to ({lat}, {lon}): {distance:.2f}")

```

This example demonstrates how loops can help you perform the same calculation (computing distance) for multiple coordinate pairs without writing repetitive code. The `:.2f` formatting ensures the distance is displayed with two decimal places. Note that we created a function named `calculate_distance` to encapsulate the distance calculation logic. We will learn more about functions in the **Functions and Classes** chapter.

11.3.3. Iterating Through Geographic Collections

For loops work with many different types of collections. Here are some common patterns you'll encounter in geospatial work:

```

# Working with a list of place names
cities = ["New York", "Los Angeles", "Chicago", "Houston", "Phoenix"]

print("Major US Cities:")
for i, city in enumerate(cities):
    print(f"{i+1}. {city}")

```


In the above example, we used a for loop to iterate through a list of city names. The `enumerate` function is used to get the index of each item in the list. This allows us to print the index and the city name in a single line. The index starts at 0 and increments by 1 for each item in the list. This can be useful when you need to keep track of the position of each item in the list.

11.4. While Loops

While loops provide a different approach to repetition. They continue executing a block of code as long as a specified condition remains true. Unlike for loops, which iterate through a known sequence, while loops are useful when you don't know in advance how many iterations you'll need.

In geospatial programming, while loops are particularly useful for processing data until certain conditions are met, such as reading GPS points until you reach a specific location, processing elevation data until you find a peak, or continuing calculations until you achieve a desired level of accuracy.

11.4.1. Basic While Loop Structure

A while loop requires a condition that can be evaluated as `True` or `False`. The loop continues as long as the condition is `True` and stops when it becomes `False`:

```
counter = 0
while counter < len(coordinates):
    lat, lon = coordinates[counter]
    print(f"Processing coordinate: ({lat}, {lon})")
    counter += 1
```

Important Note: When using while loops, always make sure your condition will eventually become `False`, or you'll create an infinite loop! In the example above, we increment the `counter` variable in each iteration to ensure the loop will eventually end.

11.4.2. Practical While Loop Example

Here's a more practical example of using a while loop to process data until a condition is met:

```
# Simulate searching for a coordinate within a certain range
import random

target_latitude = 40.0
tolerance = 0.1
attempts = 0
max_attempts = 10

print(f"Searching for coordinates near latitude {target_latitude}")

while attempts < max_attempts:
    # Simulate getting a new coordinate
    random_lat = random.uniform(39.5, 40.5)
    random_lon = random.uniform(-74.5, -73.5)
```

```

    attempts += 1

    print(f"Attempt {attempts}: Found coordinate ({random_lat:.4f},
    {random_lon:.4f})")

    # Check if we're close enough to the target
    if abs(random_lat - target_latitude) <= tolerance:
        print(f"✓ Found coordinate within tolerance after {attempts} attempts!")
        break
    else:
        print(f"✗ Could not find suitable coordinate within {max_attempts} attempts")

```

The above example shows how to generate random coordinates until you find one within a certain tolerance of a target latitude. This is a common pattern in geospatial programming when you need to find a specific location in a dataset. Try running the code block multiple times to see how the results vary.

11.5. Control Statements: Making Decisions in Your Code

Control statements allow your programs to make decisions and execute different blocks of code based on specific conditions. They are essential for creating intelligent, responsive programs that can handle different situations appropriately.

In geospatial programming, control statements help you categorize data, filter information, handle different coordinate systems, validate input data, and respond to various geographic conditions. Learning to use these statements effectively will make your programs more robust and versatile.

11.5.1. The if Statement

The `if` statement is the most basic form of conditional logic. It allows you to execute code only when a specific condition is met:

```

for lat, lon in coordinates:
    if lat > 0:
        print(f"{lat} is in the Northern Hemisphere")
    elif lat < 0:
        print(f"{lat} is in the Southern Hemisphere")
    else:
        print(f"{lat} is near the equator")

```

This example demonstrates all three types of conditional statements:

- `if` : Checks the first condition (latitude > 0)
- `elif` : Checks an alternative condition if the first one is false (latitude < 0)
- `else` : Executes if none of the previous conditions are true (latitude = 0)

11.5.2. Multiple Conditions and Complex Logic

You can create more sophisticated decision-making logic by combining multiple conditions:

```

for lat, lon in coordinates:
    if lat > 0:
        hemisphere = "Northern"
    else:
        hemisphere = "Southern"

    if lon > 0:
        direction = "Eastern"
    else:
        direction = "Western"

    print(
        f"The coordinate ({lat}, {lon}) is in the {hemisphere} Hemisphere and
        {direction} Hemisphere."
    )

```

The above example combines a for loop with an if statement to categorize coordinates by hemisphere. This demonstrates how loops and control statements can be used together to process and analyze geospatial data.

11.5.3. Logical Operators for Complex Conditions

You can use logical operators (`and` , `or` , `not`) to create more complex conditions:

```

# Classify coordinates by quadrant
for lat, lon in coordinates:
    if lat > 0 and lon > 0:
        quadrant = "Northeast"
    elif lat > 0 and lon < 0:
        quadrant = "Northwest"
    elif lat < 0 and lon > 0:
        quadrant = "Southeast"
    else: # lat < 0 and lon < 0
        quadrant = "Southwest"

    print(f"Coordinate ({lat}, {lon}) is in the {quadrant} quadrant")

```

The `and` and `or` operators are used to combine multiple conditions in a single expression. The `and` operator returns `True` if both conditions are `True`, otherwise it returns `False`. The `or` operator returns `True` if at least one of the conditions is `True`, otherwise it returns `False`. The `not` operator returns the opposite of the condition.

11.6. Combining Loops and Control Statements

The real power of loops and control statements becomes apparent when you combine them to create sophisticated data processing workflows. This combination allows you to iterate through datasets while making intelligent decisions about each piece of data you encounter.

Common patterns include filtering data based on conditions, counting items that meet specific criteria, transforming data selectively, and building new datasets from existing ones.

```
filtered_coordinates = []
for lat, lon in coordinates:
    if lon > 0:
        filtered_coordinates.append((lat, lon))
print(f"Filtered coordinates (only with positive longitude):
{filtered_coordinates}")
```

11.6.1. Counting and Analyzing Data

Loops combined with control statements are excellent for analyzing patterns in your data:

```
southern_count = 0
for lat, lon in coordinates:
    if lat < 0:
        southern_count += 1
print(f"Number of coordinates in the Southern Hemisphere: {southern_count}")
```

11.6.2. Building Summary Statistics

Here's a more comprehensive example that demonstrates multiple analysis techniques. Given a list of coordinates, we will count the number of valid coordinates, and then count the number of coordinates in each hemisphere and longitude quadrant. In a real-world application, you might use this type of analysis to understand the distribution of data points across different geographic regions.

```
# Analyze a set of coordinates
analysis_coordinates = [
    (40.7128, -74.0060), # New York
    (-33.8688, 151.2093), # Sydney
    (51.5074, -0.1278), # London
    (-1.2921, 36.8219), # Nairobi
    (35.6762, 139.6503), # Tokyo
]

# Initialize counters
northern_count = 0
southern_count = 0
eastern_count = 0
western_count = 0
valid_coordinates = []

print("Coordinate Analysis:")
print("-" * 40)
```

```

for lat, lon in analysis_coordinates:
    # Validate coordinates (basic check)
    if -90 <= lat <= 90 and -180 <= lon <= 180:
        valid_coordinates.append((lat, lon))

    # Count by hemisphere
    if lat >= 0:
        northern_count += 1
    else:
        southern_count += 1

    # Count by longitude
    if lon >= 0:
        eastern_count += 1
    else:
        western_count += 1

    print(f"Valid: ({lat:7.4f}, {lon:8.4f})")
else:
    print(f"Invalid: ({lat}, {lon}) - coordinates out of range")

print(f"\nSummary:")
print(f"Valid coordinates: {len(valid_coordinates)}")
print(f"Northern Hemisphere: {northern_count}")
print(f"Southern Hemisphere: {southern_count}")
print(f"Eastern Longitude: {eastern_count}")
print(f"Western Longitude: {western_count}")

```

11.7. Loop and Control Statement Decision Guide

Understanding when to use different types of loops and control statements will make your geospatial programming more effective:

Use For Loops When:

- You know the sequence of items you want to process
- You need to iterate through lists, tuples, or other collections
- You want to process each item in a dataset exactly once
- You're working with coordinate pairs, feature lists, or data tables

Use While Loops When:

- You need to continue processing until a condition is met
- You don't know in advance how many iterations you'll need
- You're searching for something or waiting for a condition to change
- You need to process data until you reach a threshold or target

Use Control Statements When:

- You need to make decisions based on data values
- You want to categorize or classify geographic data

- You need to filter data based on specific criteria
- You want to handle different types of input data appropriately

11.8. Key Takeaways

The key takeaways from this chapter are:

- **Loops eliminate repetition:** Instead of writing the same code multiple times, loops allow you to process multiple data points efficiently with just a few lines of code.
- **Control statements add intelligence:** They enable your programs to make decisions and respond appropriately to different data conditions, making your code more robust and versatile.
- **Combination creates power:** When you combine loops with control statements, you can create sophisticated data processing workflows that filter, analyze, and transform geospatial data effectively.
- **Practice with real data:** The best way to master these concepts is to apply them to actual geospatial datasets. Start with simple coordinate lists and gradually work with more complex geographic data.
- **Plan your logic:** Before writing complex loops with multiple conditions, sketch out your logic flow to ensure your program will handle all possible cases correctly.

These fundamental concepts will serve as building blocks for more advanced geospatial programming techniques you'll learn in subsequent chapters.

11.9. Exercises

11.9.1. Exercise 1: Using For Loops to Process Coordinate Lists

- Create a list of tuples representing coordinates (latitude, longitude).
- Write a `for` loop that prints each coordinate and indicates whether it is in the Northern or Southern Hemisphere based on the latitude.

11.9.2. Exercise 2: While Loops for Iterative Processing

- Create a list of coordinates (latitude, longitude).
- Write a `while` loop that continues to print each coordinate until it encounters a coordinate with a negative latitude.
- Stop the loop once this condition is met.

11.9.3. Exercise 3: Conditional Logic in Loops

- Create a list of coordinates and use a `for` loop to iterate over them.
- Use an `if-elif-else` statement inside the loop to classify each coordinate based on its longitude:
 - Print "Eastern Hemisphere" if the longitude is greater than 0.
 - Print "Western Hemisphere" if the longitude is less than 0.

11.9.4. Exercise 4: Filtering Data with Combined Loops and Conditionals

- Given a list of coordinates, filter out and store only those located in the Southern Hemisphere (latitude < 0).
- Count the number of coordinates that meet this condition and print the result.

11.9.5. Exercise 5: Generating and Analyzing Random Coordinates

- Write a program that generates random coordinates (latitude between $[-90, 90]$ degrees and longitude between $[-180, 180]$ degrees).
- Use a `while` loop to keep generating coordinates until a pair with both latitude and longitude greater than 50 is generated.
- Print each generated coordinate and the final coordinate that meets the condition.

Chapter 12. Functions and Classes

12.1. Introduction

Functions and classes are two of the most powerful organizational tools in Python programming, and they become especially valuable when working with geospatial data. These concepts allow you to structure your code in ways that make it more readable, reusable, and maintainable—essential qualities when dealing with complex spatial analysis tasks.

Functions serve as the building blocks of organized code. Instead of writing the same distance calculation or coordinate transformation code multiple times throughout your program, you can encapsulate that logic into a function and call it whenever needed. This approach, known as “Don’t Repeat Yourself” (DRY), makes your code more efficient and reduces the chance of errors.

Classes take organization a step further by allowing you to bundle related data and functions together into logical units. For example, instead of managing separate variables for latitude, longitude, elevation, and place name, you can create a `Location` class that keeps all this information together along with methods to manipulate it.

In geospatial programming, these concepts become particularly powerful. You might create functions to calculate distances, convert between coordinate systems, or validate GPS data. You could design classes to represent geographic features like points, lines, or polygons, each with their own properties and behaviors. As your spatial analysis projects grow more complex, these organizational tools will help you build sophisticated yet manageable geospatial applications.

12.2. Learning Objectives

By the end of this chapter, you should be able to:

- Define and use functions to perform specific tasks and promote code reuse in geospatial applications
- Understand and implement classes to represent complex geospatial data structures, such as geographic features
- Combine functions and classes to create modular and scalable geospatial tools
- Apply object-oriented programming principles to organize and manage geospatial data and operations effectively
- Develop the skills to extend existing classes and create new ones tailored to specific geospatial tasks

12.3. Functions: Building Reusable Code Blocks

Functions are fundamental building blocks that allow you to package code into reusable, named units. Think of functions as specialized tools in a toolbox—each one designed to perform a specific task that you can use whenever needed. In geospatial programming, functions help you avoid writing the same coordinate calculations, data transformations, or validation routines repeatedly.

The power of functions lies in their ability to take inputs (called parameters), process them in some way, and return results. This creates a clear interface between different parts of your program and makes your code more modular and easier to test and maintain.

12.3.1. Defining a Simple Function

The basic structure of a function includes the `def` keyword, a function name, parameters in parentheses, and a colon followed by an indented code block. Here's a simple function that adds two numbers:

```
def add(a, b):  
    return a + b  
  
# Example usage  
result = add(5, 3)  
print(f"Result: {result}")
```

This function demonstrates the key components of function definition:

- `def add(a, b):` - The function definition with name and parameters
- `return a + b` - The operation performed and value returned
- `add(5, 3)` - How to call the function with specific arguments

The `return` statement is crucial. It sends a value back to the code that called the function, allowing you to capture and use the result.

12.3.2. Parameters with Default Values

Default parameters make functions more flexible by providing fallback values when certain arguments aren't provided. This is particularly useful in geospatial applications where you might have standard units, default coordinate systems, or common calculation parameters that don't need to be specified every time.

```
def greet(name, greeting="Hello"):  
    return f"{greeting}, {name}!"  
  
# Example usage  
print(greet("Alice")) # Uses the default greeting  
print(greet("Bob", "Hi")) # Overrides the default greeting
```

This example shows how default parameters work:

- When calling `greet("Alice")`, only the name is provided, so Python uses the default greeting "Hello"
- When calling `greet("Bob", "Hi")`, both parameters are provided, so the default is overridden

Default parameters must come after required parameters in the function definition. This pattern allows you to create functions that are both simple to use in common cases and flexible when needed.

12.3.3. Understanding Function Calls

Function calls are how you actually use the functions you've defined. The process involves providing the necessary arguments (inputs) and capturing the returned value (output). Understanding this flow is essential for building more complex programs.

```
# Function to multiply two numbers
def multiply(a, b):
    return a * b

# Calling the function
result = multiply(4, 5)
print(f"Multiplication Result: {result}")
```

Notice the pattern here:

1. **Function definition:** We define what the function does with `def multiply(a, b):`
2. **Function call:** We invoke the function with specific values `multiply(4, 5)`
3. **Result capture:** We store the returned value in a variable `result = multiply(4, 5)`
4. **Result usage:** We use the returned value in other operations

This pattern of defining, calling, and using results forms the foundation of modular programming.

12.3.4. Geospatial Example: Haversine Function

Now let's apply function concepts to a real geospatial problem. The [Haversine formula](#)³⁶ calculates the distance between two points on the Earth's surface (see [Figure 8](#)).

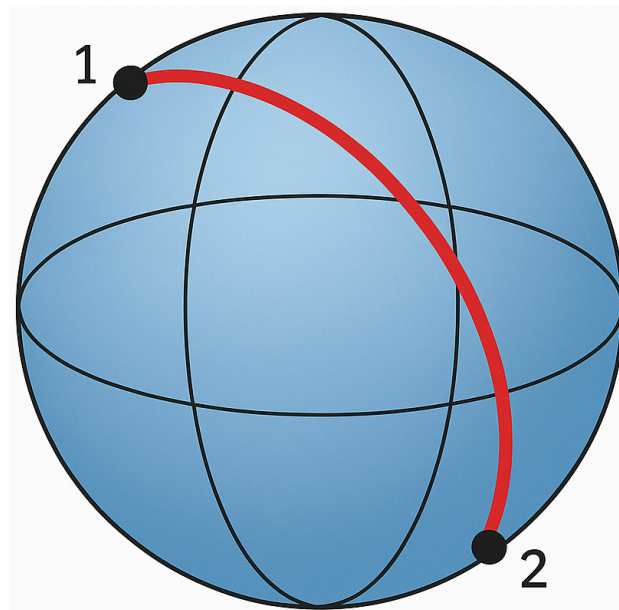


Figure 8: A diagram illustrating great-circle distance (drawn in red) between two points on a sphere. First, let's import the necessary functions from the `math` module:

```
from math import radians, sin, cos, sqrt, atan2
```

³⁶<https://shorturl.at/rO0Yy>

Then, we define the function `haversine` that takes four parameters: `lat1`, `lon1`, `lat2`, and `lon2`. The function returns the distance between the two points in kilometers.

```
def haversine(lat1, lon1, lat2, lon2):
    R = 6371.0 # Earth radius in kilometers
    dlat = radians(lat2 - lat1)
    dlon = radians(lon2 - lon1)
    a = (
        sin(dlat / 2) ** 2
        + cos(radians(lat1)) * cos(radians(lat2)) * sin(dlon / 2) ** 2
    )
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    distance = R * c
    return distance

# Example usage
distance = haversine(35.6895, 139.6917, 34.0522, -118.2437)
print(f"Distance: {distance:.2f} km")
```

This function demonstrates several important concepts:

- **Meaningful parameters:** `lat1`, `lon1`, `lat2`, `lon2` clearly indicate what inputs are expected
- **Mathematical operations:** The function encapsulates complex trigonometric calculations
- **Consistent output:** Always returns distance in the same units (kilometers)
- **Reusability:** Can be used wherever distance calculations are needed

Try the function with different coordinate values to see how it works.

12.3.5. Function with Default Values and Geospatial Application

Now let's enhance the `haversine` function to demonstrate default parameters in action. We'll modify it to accept an optional Earth radius parameter, which defaults to kilometers but can be changed for other units like miles:

```
def haversine(lat1, lon1, lat2, lon2, radius=6371.0):
    dlat = radians(lat2 - lat1)
    dlon = radians(lon2 - lon1)
    a = (
        sin(dlat / 2) ** 2
        + cos(radians(lat1)) * cos(radians(lat2)) * sin(dlon / 2) ** 2
    )
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    distance = radius * c
    return distance

# Example usage in kilometers
distance_km = haversine(35.6895, 139.6917, 34.0522, -118.2437)
```

```
print(f"Distance in kilometers: {distance_km:.2f} km")

# Example usage in miles (radius of Earth is approximately 3958.8 miles)
distance_miles = haversine(35.6895, 139.6917, 34.0522, -118.2437, radius=3958.8)
print(f"Distance in miles: {distance_miles:.2f} miles")
```

This enhanced version shows the power of default parameters in geospatial programming:

- **Convenience:** Most of the time, you want distance in kilometers, so the default works perfectly
- **Flexibility:** When you need different units, you can easily override the default
- **Professional practice:** Real geospatial applications often need to handle multiple unit systems

12.3.6. Processing Multiple Coordinates

Functions become even more powerful when they can handle collections of data. Here's how you can create a function that processes multiple coordinate pairs efficiently:

```
def batch_haversine(coord_list):
    distances = []
    for i in range(len(coord_list) - 1):
        lat1, lon1 = coord_list[i]
        lat2, lon2 = coord_list[i + 1]
        distance = haversine(lat1, lon1, lat2, lon2)
        distances.append(distance)
    return distances

# Example usage
coordinates = [(35.6895, 139.6917), (34.0522, -118.2437), (40.7128, -74.0060)]
distances = batch_haversine(coordinates)
print(f"Distances: {distances}")
```

This function demonstrates a common pattern in geospatial programming: processing sequences of coordinates to calculate cumulative statistics, analyze routes, or track movement patterns.

12.3.7. Advanced Parameter Handling

Python provides powerful ways to handle different types of function parameters. Understanding these patterns will make your geospatial functions more flexible and user-friendly.

12.3.7.1. Variable Arguments with *args

You can create functions that accept any number of arguments using `*args`:

```
def average(*numbers):
    return sum(numbers) / len(numbers)
```

```
# Example usage
print(average(10, 20, 30)) # 20.0
print(average(5, 15, 25, 35)) # 20.0
```

The `*args` parameter allows the function to accept any number of arguments, which are then treated as a tuple inside the function. This is useful for calculations where the number of inputs might vary.

12.3.7.2. Keyword Arguments with `**kwargs`

You can also use `**kwargs` (keyword arguments) to accept a variable number of named parameters. This is particularly useful for geospatial functions that might need to handle optional metadata or attributes:

```
def describe_point(latitude, longitude, **kwargs):
    description = f"Point at ({latitude}, {longitude})"

    # Add optional keyword arguments to the description
    for key, value in kwargs.items():
        description += f", {key}: {value}"

    return description

# Example usage
print(describe_point(35.6895, 139.6917, name="Tokyo", population=37400000))
print(describe_point(34.0522, -118.2437, name="Los Angeles", state="California"))
```

The output should look like this:

```
Point at (35.6895, 139.6917), name: Tokyo, population: 37400000
Point at (34.0522, -118.2437), name: Los Angeles, state: California
```

This pattern allows functions to handle a flexible set of optional parameters, making them more adaptable to different use cases while maintaining a clean, consistent interface.

12.4. Classes: Organizing Data and Behavior Together

Classes represent the next level of code organization beyond functions. While functions group related operations, classes group related data and the operations that work on that data into single, cohesive units called objects. In geospatial programming, classes are invaluable for representing complex spatial entities and their behaviors.

Think of classes as blueprints or templates. Just as an architectural blueprint defines how to build a house, a class defines how to create objects that share common characteristics and behaviors. For example, you might create a `Point` class that defines how all geographic points should be structured and what operations they can perform.

12.4.1. Defining a Simple Class

Let's start with a simple `Point` class that demonstrates the basic structure of class definition. This class will represent geographic points with coordinates and optional names:

```
class Point:
    def __init__(self, latitude, longitude, name=None):
        self.latitude = latitude
        self.longitude = longitude
        self.name = name

    def __str__(self):
        return f"{self.name or 'Point'} ({self.latitude}, {self.longitude})"

# Example usage
point1 = Point(35.6895, 139.6917, "Tokyo")
print(point1)
```

This class demonstrates several key concepts:

- **`__init__` method:** This special method (called a constructor) runs automatically when you create a new object, setting up the initial state
- **`self` parameter:** Refers to the specific instance of the class being created or operated on
- **Instance attributes:** `self.latitude`, `self.longitude`, and `self.name` store data for each point
- **`__str__` method:** Another special method that defines how the object appears when printed

12.4.2. Adding Methods to a Class

Methods are functions defined inside a class that operate on the class's data. They allow objects to perform actions using their own attributes. Let's add a method to calculate distances between points:

```
class Point:
    def __init__(self, latitude, longitude, name=None):
        self.latitude = latitude
        self.longitude = longitude
        self.name = name

    def distance_to(self, other_point):
        return haversine(
            self.latitude, self.longitude, other_point.latitude,
            other_point.longitude
        )

# Example usage
point1 = Point(35.6895, 139.6917, "Tokyo")
point2 = Point(34.0522, -118.2437, "Los Angeles")
print(
```

```
f"Distance from {point1.name} to {point2.name}:
{point1.distance_to(point2):.2f} km"
)
```

12.4.3. Constructor with Default Values

You can also use default values in the constructor of a class. This is useful when you want to create a class with some default values for some attributes. For example, you might want to create a `Point` class with a default name of “Unknown” if no name is provided:

```
class Point:
    def __init__(self, latitude, longitude, name="Unnamed"):
        self.latitude = latitude
        self.longitude = longitude
        self.name = name
```

12.5. Combining Functions and Classes

You can use functions within classes to create more powerful and flexible geospatial tools. For instance, by incorporating distance calculations and midpoints, we can make the `Point` class much more versatile.

Let’s create a `Route` class that contains a list of points and a method to calculate the total distance of the route.

```
class Route:
    def __init__(self, points):
        self.points = points

    def total_distance(self):
        total_dist = 0
        for i in range(len(self.points) - 1):
            total_dist += self.points[i].distance_to(self.points[i + 1])
        return total_dist

# Example usage
route = Route([point1, point2])
print(f"Total distance: {route.total_distance():.2f} km")
```

12.6. Function and Class Design Guidelines

Understanding when and how to use functions and classes effectively will make your geospatial programming more organized and maintainable:

Use Functions When:

- You need to perform the same operation multiple times with different inputs

- You want to break down complex calculations into manageable pieces
- You need to validate, transform, or process data in a consistent way
- You want to create reusable utilities for common geospatial tasks

Use Classes When:

- You need to represent complex entities with multiple related attributes
- You want to bundle data and the operations that work on that data together
- You're modeling real-world geographic objects (points, routes, regions)
- You need to maintain state and behavior together in a logical unit

Best Practices:

- Give functions and classes descriptive names that clearly indicate their purpose
- Keep functions focused on a single task (single responsibility principle)
- Use default parameters to make functions more flexible and user-friendly
- Test your functions with various inputs to ensure they work correctly

12.7. Key Takeaways

The following are some key takeaways from this chapter:

- **Functions eliminate code duplication:** By encapsulating common operations into reusable functions, you make your code more efficient and easier to maintain.
- **Classes organize complexity:** When you have related data and operations, classes provide a natural way to keep them together and create more intuitive code structures.
- **Parameters add flexibility:** Default parameters, `*args`, and `**kwargs` allow you to create functions that are both simple to use and highly flexible.
- **Geospatial applications benefit greatly:** Distance calculations, coordinate transformations, and spatial data management are perfect candidates for function and class organization.
- **Start simple, then expand:** Begin with basic functions and classes, then add features and complexity as your understanding and requirements grow.

These organizational tools will serve as the foundation for more advanced geospatial programming techniques you'll encounter in later chapters.

12.8. Exercises

12.8.1. Exercise 1: Calculating Distances with Functions

- Define a function `calculate_distance` that takes two geographic coordinates (latitude and longitude) and returns the distance between them using the [Haversine formula](https://shorturl.at/rO0Yy)³⁷.
- Use this function to calculate the distance between multiple pairs of coordinates.
- Test your function with coordinates for cities you know and verify the results make sense.

³⁷<https://shorturl.at/rO0Yy>

12.8.2. Exercise 2: Batch Distance Calculation

- Create a function `batch_distance_calculation` that accepts a list of coordinate pairs and returns a list of distances between consecutive pairs.
- Test the function with a list of coordinates representing several cities.
- Add an optional parameter to specify the units (kilometers or miles) with a default value.

12.8.3. Exercise 3: Creating and Using a Point Class

- Define a `Point` class to represent a geographic point with attributes `latitude`, `longitude`, and `name`.
- Add a method `distance_to` that calculates the distance from one point to another.
- Include a `__str__` method that provides a readable representation of the point.
- Instantiate several `Point` objects and calculate the distance between them.
- Create a method `is_in_northern_hemisphere` that returns True if the point is north of the equator.

Chapter 13. Working with Files

13.1. Introduction

In the world of geospatial programming, data rarely exists in isolation within your program. Most often, it comes from external sources: text files containing GPS coordinates, CSV files with spatial measurements, or configuration files with map settings. Learning to work with files is therefore fundamental to any geospatial application.

This chapter introduces the essential techniques for reading from and writing to files in Python, with a special focus on handling geospatial data. You'll also learn about exception handling—a critical skill that ensures your programs can gracefully handle unexpected situations like missing files, corrupted data, or network interruptions.

Think of file handling as the bridge between your Python program and the external world of data. Whether you're processing GPS tracks from a hiking expedition, analyzing weather station data, or working with coordinates from a mapping survey, you'll need these fundamental skills to get data into your program and save results back out.

Exception handling, meanwhile, is your safety net. In real-world scenarios, files might be missing, data might be corrupted, or network connections might fail. Rather than letting these issues crash your program, exception handling allows you to anticipate problems and respond appropriately—perhaps by displaying a helpful error message, trying an alternative approach, or gracefully skipping problematic data.

13.2. Learning Objectives

By the end of this chapter, you should be able to:

- Read from and write to files in Python, with a particular focus on handling geospatial data
- Implement exception handling using `try`, `except`, and `finally` blocks to manage errors that may occur during file operations
- Combine file handling and exception handling to create robust and reliable geospatial applications
- Develop the skills to identify and manage common issues in file processing, such as missing files, corrupt data, or formatting errors
- Ensure that your geospatial programs can handle real-world data scenarios effectively by using best practices for file and exception handling

13.3. Creating a Sample File

Before diving into file reading and writing, we need some data to work with. In real-world geospatial projects, you might receive coordinate data from GPS devices, surveys, or online sources. To simulate this scenario, we'll create a sample file containing coordinates from major cities around the world. Each line contains latitude and longitude separated by a comma. These coordinates represent: Tokyo, Los Angeles, London, Sydney, and Paris.

This approach demonstrates an important principle: always ensure your data files exist before trying to process them. Creating sample data programmatically is also useful for testing your programs and providing examples for others to follow.

```

sample_data = """35.6895,139.6917
34.0522,-118.2437
51.5074,-0.1278
-33.8688,151.2093
48.8566,2.3522"""

output_file = "coordinates.txt"

try:
    with open(output_file, "w") as file:
        file.write(sample_data)
    print(f"Sample file '{output_file}' has been created successfully.")
except Exception as e:
    print(f"An error occurred while creating the file: {e}")

```

This code example demonstrates several important file handling concepts:

1. **String literals with triple quotes:** The `sample_data` uses triple quotes (`"""`) to create a multi-line string, which is perfect for storing structured data like coordinates.
2. **The `with` statement:** This is Python’s recommended way to work with files. It automatically handles file opening and closing, even if an error occurs. This prevents resource leaks and ensures data is properly saved.
3. **Exception handling:** The `try-except` block catches any errors that might occur during file creation, such as permission issues or disk space problems.

After running this code, you’ll have a `coordinates.txt` file in your current directory containing five coordinate pairs. This file will serve as our sample data for the upcoming examples. The coordinates represent major cities: Tokyo, Los Angeles, London, Sydney, and Paris—giving us a nice global distribution for testing.

Note that the `write` method is used to write the string to the file. The `w` mode is used to open the file for writing. If the file does not exist, it will be created. If the file exists, it will be overwritten. If you do not want to overwrite the file, you can use the `a` mode to append to the file.

13.4. Reading and Writing Files

Now that we have our sample data, let’s explore the fundamental operations of reading from and writing to files. These are essential skills you’ll use constantly in geospatial programming—whether you’re processing GPS tracks, importing survey data, or exporting analysis results.

Python’s file handling capabilities are built into the language, meaning you don’t need to import any special libraries for basic file operations. The key concepts we’ll cover are:

- **Reading files:** Getting data from external files into your program
- **Writing files:** Saving data from your program to external files
- **File modes:** Understanding when to use “read” vs “write” mode
- **Processing data:** Transforming data as you read or write it

Let’s start with a practical example that reads our coordinate file and creates a more readable version with formatted output.

```

# Example of reading coordinates from a file and writing to another file
input_file = "coordinates.txt"
output_file = "output_coordinates.txt"

try:
    # Step 1: Read the coordinate data from our input file
    with open(input_file, "r") as infile:
        # readlines() returns a list where each element is a line from the file
        # Each line still includes the newline character (\n) at the end
        coordinates = infile.readlines()

    # Step 2: Process the data and write it to a new file with better formatting
    with open(output_file, "w") as outfile:
        for line in coordinates:
            # strip() removes whitespace (including \n) from both ends of the
            string
            # split(",") breaks the line into parts wherever it finds a comma
            lat, lon = line.strip().split(",")

            # Write the formatted data to our output file
            # The \n adds a newline character so each coordinate is on its own
            line
            outfile.write(f"Latitude: {lat}, Longitude: {lon}\n")

        print(f"Coordinates have been written to {output_file}")

except FileNotFoundError:
    print(f"Error: The file {input_file} was not found.")
    print(
        "Make sure you ran the previous code cell to create the coordinates.txt
        file."
    )

```

This code example demonstrates how to read data from a file and write to another file. The `readlines` method is used to read the lines from the file. The `write` method is used to write the lines to the new file. The `strip` method is used to remove the newline character from the end of the line. The `split` method is used to split the line into a list of strings. You can also use the `writelines` method to write a list of strings to a file. By combining these methods, we can read the data from the file and write it to the new file with the desired format. This is a common pattern in geospatial programming when you need to process data from a file and write the results to another file.

13.5. Exception Handling

In the real world, things don't always go as planned. Files might be missing, data might be corrupted, or networks might be down. Exception handling is Python's way of dealing with these unexpected situations gracefully, rather than letting your program crash with an error.

Think of exceptions as Python's way of saying "Something went wrong, but I'll let you decide how to handle it." Without exception handling, a single missing file could stop your entire geospatial analysis. With proper exception handling, your program can continue processing other files, log the error, or try alternative approaches.

The basic structure uses three keywords:

- **try** : Contains the code that might cause an error
- **except** : Contains the code that runs if an error occurs
- **finally** : Contains code that runs regardless of whether an error occurred (optional)

Let's see how this works with a practical example of parsing coordinate data that might be malformed.

```
# Example of exception handling when parsing coordinates
def parse_coordinates(line):
    """
    Parse a line of text into latitude and longitude coordinates.

    Args:
        line (str): A string containing coordinates in the format "lat,lon"

    Returns:
        tuple: (latitude, longitude) as floats, or None if parsing fails
    """
    try:
        # Attempt to split the line and convert to numbers
        lat, lon = line.strip().split(",")
        lat = float(lat) # This might raise ValueError if lat isn't a valid
number
        lon = float(lon) # This might raise ValueError if lon isn't a valid
number
        return lat, lon

    except ValueError as e:
        # This happens when we can't convert to float or split doesn't work as
expected
        print(f>Data format error: {e}. Could not parse line: '{line.strip()}'")
        return None

    except Exception as e:
        # This catches any other unexpected errors
        print(f>An unexpected error occurred: {e}")
        return None

# Test with both valid and invalid data
test_lines = [
    "35.6895,139.6917", # Valid coordinates (Tokyo)
    "invalid data", # Invalid format
    "45.0,-119.0", # Valid coordinates
```

```

    "45.0,not_a_number", # Invalid longitude
    "only_one_value", # Missing comma
]

print("Testing coordinate parsing:")
for line in test_lines:
    coordinates = parse_coordinates(line)
    if coordinates:
        print(f"✓ Successfully parsed: {coordinates}")
    else:
        print(f"✗ Failed to parse: '{line}'")

```

This code example demonstrates how to process geographic coordinates and identify invalid data. This is a very common task in geospatial programming when you need to process field survey data or GPS data, which might contain invalid entries.

13.6. Combining File Handling and Exception Handling

Now we'll bring together everything we've learned about files and exceptions to create truly robust geospatial programs. In real-world scenarios, you need to handle multiple potential problems: files might not exist, data might be corrupted, or individual lines might have formatting issues.

The key principle is **graceful degradation** - your program should handle as much good data as possible, skip problematic data with appropriate warnings, and continue processing. This is especially important when dealing with large datasets where a few bad records shouldn't stop the entire analysis.

Let's create a function that demonstrates these principles by processing our coordinate file while handling various potential issues.

```

# Example of robust file handling with exceptions
def process_geospatial_file(input_file):
    """
    Process a file containing coordinate data, handling various potential errors.

    This function demonstrates robust file processing by:
    - Handling missing files gracefully
    - Continuing processing even when individual lines fail
    - Providing informative feedback about the processing results
    """
    processed_count = 0
    error_count = 0

    try:
        print(f"Starting to process file: {input_file}")

        with open(input_file, "r") as infile:
            for line_number, line in enumerate(infile, 1):
                # Skip empty lines

```

```

        if not line.strip():
            continue

        coordinates = parse_coordinates(line)
        if coordinates:
            lat, lon = coordinates
            print(
                f"Line {line_number}: Processed coordinates ({lat:.4f},
{lon:.4f})"
            )
            processed_count += 1
        else:
            print(f"Line {line_number}: Skipped due to parsing error")
            error_count += 1

    except FileNotFoundError:
        print(f"Error: The file '{input_file}' was not found.")
        print("Please check the file path and make sure the file exists.")
        return

    except PermissionError:
        print(f"Error: Permission denied when trying to read '{input_file}'.")
        print("Please check if you have read permissions for this file.")
        return

    except Exception as e:
        print(f"An unexpected error occurred while processing the file: {e}")
        return

    finally:
        # This block always runs, whether there was an error or not
        print(f"\n--- Processing Summary ---")
        print(f"Successfully processed: {processed_count} coordinates")
        print(f"Errors encountered: {error_count} lines")
        print(f"Finished processing {input_file}")

# Example usage with our coordinates file
process_geospatial_file("coordinates.txt")

```

13.7. Working with Different File Formats

So far we've worked with simple text files, but geospatial data often comes in various formats. While we'll explore specialized geospatial formats in later chapters, it's useful to understand how to handle common structured text formats like CSV (Comma-Separated Values).

CSV files are particularly common in geospatial work because they can easily store tabular data like coordinate lists, attribute tables, or measurement records. Let's create and work with a more structured coordinate file that includes city names.

```
# Create a CSV-style file with city names and coordinates
csv_data = """City,Latitude,Longitude
Tokyo,35.6895,139.6917
Los Angeles,34.0522,-118.2437
London,51.5074,-0.1278
Sydney,-33.8688,151.2093
Paris,48.8566,2.3522"""

csv_file = "cities.csv"

try:
    with open(csv_file, "w") as file:
        file.write(csv_data)
    print(f"CSV file '{csv_file}' created successfully.")
except Exception as e:
    print(f"Error creating CSV file: {e}")
```

Now let's read and process this structured data:

```
def read_city_coordinates(filename):
    """
    Read city coordinate data from a CSV-style file.

    Returns a list of dictionaries containing city information.
    """
    cities = []

    try:
        with open(filename, "r") as file:
            lines = file.readlines()

            # Skip the header line (first line)
            for line_num, line in enumerate(lines[1:], 2): # Start from line 2
                try:
                    # Parse each line
                    parts = line.strip().split(",")
                    if len(parts) == 3:
                        city_name = parts[0]
                        latitude = float(parts[1])
                        longitude = float(parts[2])

                        cities.append(
                            {
                                "name": city_name,
```



```

        "latitude": latitude,
        "longitude": longitude,
    }
)

except ValueError as e:
    print(f"Warning: Could not parse line {line_num}:
{line.strip()}")
    continue

except FileNotFoundError:
    print(f"Error: File '{filename}' not found.")
    return []
except Exception as e:
    print(f"Error reading file: {e}")
    return []

return cities

# Read and display the city data
cities = read_city_coordinates("cities.csv")

print(f"Successfully loaded {len(cities)} cities:")
for city in cities:
    print(f"- {city['name']}: ({city['latitude']:.4f}, {city['longitude']:.4f})")

```

13.8. Key Takeaways

Mastering file handling and exception handling forms the foundation for robust geospatial programming. Here are the essential concepts you should remember:

File Handling Fundamentals:

- Always use the `with` statement when working with files - it ensures proper resource management
- Understand the difference between read ("r"), write ("w"), and append ("a") modes
- Remember that `readlines()` preserves newline characters, so use `strip()` when processing
- Structure your data consistently (like CSV format) to make parsing easier

Exception Handling Best Practices:

- Use `try-except` blocks to handle predictable errors gracefully
- Be specific with exception types (`FileNotFoundError` , `ValueError`) when possible
- Always provide helpful error messages that guide users toward solutions
- Use the `finally` block for cleanup code that must run regardless of errors

Real-World Application:

- Design your programs to handle partial failures gracefully (skip bad data, continue processing)
- Provide informative feedback about what succeeded and what failed
- Consider edge cases like empty files, malformed data, and permission issues

- Keep track of processing statistics to help users understand the results

These skills will serve you well as we move into more advanced geospatial topics, where data quality and error handling become even more critical.

13.9. Exercises

These exercises will help you practice the fundamental file handling and exception handling concepts covered in this chapter. Each exercise builds on the previous ones, so it's recommended to complete them in order.

13.9.1. Exercise 1: Reading and Writing Files

Create two complementary functions that demonstrate basic file I/O operations:

1. **`read_coordinates(filename)`** : This function should read a file containing coordinates (one per line, in "lat,lon" format) and return them as a list of tuples. Handle cases where the file doesn't exist or contains malformed data.
2. **`write_coordinates(coordinates, filename)`** : This function should take a list of coordinate tuples and write them to a file, one coordinate pair per line. Include proper exception handling for write permissions and disk space issues.

Testing your functions:

- Create a test file with some coordinates
- Read them using your function
- Write them to a new file using your function
- Verify that the process works correctly and handles errors gracefully

13.9.2. Exercise 2: Processing Coordinates from a File

Build on your file handling skills by creating a more complex data processing function:

Task: Create a function `calculate_distances_from_file(input_file, output_file)` that:

1. Reads coordinates from a file (using concepts from previous chapters about the `Point` class)
2. Calculates the distance between each consecutive pair of points
3. Writes the results to a new file with descriptive formatting

Requirements:

- Handle file-related exceptions (missing files, permission errors)
- Skip malformed lines while continuing to process valid data
- Provide a summary of how many distances were calculated and how many lines had errors
- Format the output file with clear labels (e.g., "Distance from point 1 to point 2: 1234.56 km")

Challenge: Can you modify your function to also calculate the total distance of the entire route?

```
# Create a sample coordinates.txt file
sample_data = """35.6895,139.6917
```

```

34.0522,-118.2437
51.5074,-0.1278
-33.8688,151.2093
48.8566,2.3522"""

output_file = "coordinates.txt"

try:
    with open(output_file, "w") as file:
        file.write(sample_data)
    print(f"Sample file '{output_file}' has been created successfully.")
except Exception as e:
    print(f"An error occurred while creating the file: {e}")

```

13.9.3. Exercise 3: Exception Handling in Data Processing

Focus on building robust error handling into data processing workflows:

Task: Create or modify a batch processing function that demonstrates advanced exception handling techniques.

Scenario: You have multiple coordinate files to process, but some files might be missing, corrupted, or contain mixed data quality. Your function should:

1. **Process multiple files in a batch** (accept a list of filenames)
2. **Handle different types of errors:**
 - Missing files (continue with other files)
 - Permission errors (log and continue)
 - Data format errors (skip bad lines, continue processing)
 - Calculation errors (handle edge cases like identical coordinates)
3. **Provide comprehensive reporting:**
 - Which files were processed successfully
 - How many records were processed vs. skipped in each file
 - Summary statistics of the entire batch operation

Chapter 14. Data Analysis with NumPy and Pandas

14.1. Introduction

In geospatial programming, you'll frequently work with large amounts of numerical data: coordinates from GPS devices, elevation measurements, temperature readings, population statistics, and much more. While Python's built-in data types (lists, dictionaries) can handle small datasets, they become inefficient and cumbersome when dealing with the volume and complexity of real-world geospatial data.

This is where [NumPy](https://numpy.org)³⁸ and [Pandas](https://pandas.pydata.org)³⁹ become essential tools in your geospatial programming toolkit. Think of NumPy as your mathematical powerhouse. It provides efficient storage and operations for numerical data in arrays, making calculations on thousands or millions of data points fast and straightforward. Pandas, on the other hand, is your data organization expert. It excels at handling structured, tabular data (like spreadsheets) and provides intuitive tools for cleaning, analyzing, and manipulating datasets.

Why NumPy matters for geospatial work:

- Efficiently stores and processes large arrays of coordinates, measurements, or sensor data
- Provides fast mathematical operations needed for distance calculations, coordinate transformations, and statistical analysis
- Forms the foundation for most other geospatial Python libraries

Why Pandas matters for geospatial work:

- Handles datasets with mixed data types (city names, coordinates, population numbers, dates)
- Makes it easy to filter, group, and summarize geospatial datasets
- Seamlessly reads data from common formats like CSV files, databases, and web APIs
- Provides tools for handling missing or inconsistent data—common issues in real-world datasets

Together, these libraries will transform how you work with geospatial data, making complex analyses both possible and efficient.

14.2. Learning Objectives

By the end of this chapter, you should be able to:

- Understand the basics of `NumPy` arrays and how to perform operations on them
- Utilize `Pandas` `DataFrames` to organize, analyze, and manipulate tabular data
- Apply `NumPy` and `Pandas` in geospatial programming to process and analyze geospatial datasets
- Combine `NumPy` and `Pandas` to streamline data processing workflows
- Develop the ability to perform complex data operations, such as filtering, aggregating, and transforming geospatial data

14.3. Introduction to NumPy

NumPy (Numerical Python) is the foundation of scientific computing in Python. At its core, NumPy provides a powerful data structure called an **array**—think of it as a more efficient version of Python lists, but specifically designed for numerical data.

³⁸<https://numpy.org>

³⁹<https://pandas.pydata.org>

What makes NumPy arrays special?

- **Speed:** Operations on NumPy arrays are much faster than equivalent operations on Python lists
- **Memory efficiency:** Arrays store data more compactly, important when working with large datasets
- **Vectorization:** You can perform mathematical operations on entire arrays at once, rather than looping through individual elements
- **Multidimensional:** Arrays can have multiple dimensions, perfect for storing coordinate grids, images, or sensor data

In geospatial contexts, you'll use NumPy arrays to store:

- Lists of coordinates (latitude/longitude pairs)
- Elevation data from digital elevation models
- Time series of measurements from weather stations
- Pixel values from satellite imagery
- Results of spatial calculations

Let's start with the basics of creating and working with NumPy arrays.

14.3.1. Creating NumPy Arrays

There are several ways to create NumPy arrays, depending on your data source and needs. Understanding these different creation methods is important because in geospatial work, your data might come from various sources. Sometimes you'll create arrays from Python lists, other times you'll need arrays filled with zeros or specific patterns.

To use NumPy, you need to import it first. The convention is to import it as `np`, which is short for "NumPy".

```
import numpy as np
```

Let's create a NumPy array from a list of numbers, which may represent elevation measurements at different points:

```
elevation_readings = np.array([245, 312, 156, 423, 678])
print(f"1D Array (elevation readings): {elevation_readings}")
print(f"Data type: {elevation_readings.dtype}")
print(f"Array shape: {elevation_readings.shape}")
```

Each NumPy array is an object that contains a sequence of numbers. It is similar to a list in Python, but it is more efficient for numerical operations. The `dtype` attribute of an array tells you the data type of the elements in the array. The `shape` attribute of an array tells you the dimensions of the array.

Let's create a 2D array of coordinates:

```
# Each row represents one location: [latitude, longitude]
coordinates = np.array(
    [[35.6895, 139.6917], [40.7128, -74.0060], [51.5074, -0.1278]] # Tokyo #
    New York
) # London
```

```
print(f"2D Array (city coordinates):\n{coordinates}")
print(f"Array shape: {coordinates.shape}") # (3 rows, 2 columns)
print(f"Number of cities: {coordinates.shape[0]}")
```

Sometimes you need to create a NumPy array with a specific pattern. For example, you might want to create an array of zeros or ones. You can do this with the `zeros` and `ones` functions.

Let's create an array of zeros, which may be useful when you need to initialize an array before filling it with data:

```
# For example, you might create a grid to store calculated distances
distance_matrix = np.zeros((3, 3))
print(f"Distance matrix (initialized with zeros):\n{distance_matrix}")
print(f"This could store distances between {distance_matrix.shape[0]} cities")
```

Alternatively, you can create a NumPy array of ones:

```
# You can multiply by any number to get arrays filled with that value
weights = np.ones((2, 4)) * 0.5 # Array filled with 0.5
print(f"Array of weights (0.5):\n{weights}")

# Or create an array for marking valid data points
valid_data_flags = np.ones((5,), dtype=bool) # All True initially
print(f"Data validity flags: {valid_data_flags}")
```

The `arange` function is a built-in function that creates a NumPy array with a sequence of numbers. It is similar to the `range` function in Python, but it returns a NumPy array instead of a list.

```
# For example, creating a series of years for time series analysis
years = np.arange(2010, 2021, 1) # From 2010 to 2020, step by 1
print(f"Years array: {years}")

# Or creating latitude values for a grid
latitudes = np.arange(-90, 91, 30) # From -90 to 90 degrees, every 30 degrees
print(f"Latitude grid: {latitudes}")

# You can also use linspace for evenly spaced values
longitudes = np.linspace(-180, 180, 7) # 7 evenly spaced values from -180 to 180
print(f"Longitude grid: {longitudes}")
```

14.3.2. Basic Array Operations

One of NumPy's greatest strengths is **vectorization**—the ability to perform mathematical operations on entire arrays at once, without writing explicit loops. This is not only more convenient but also much faster than operating on individual elements.

In geospatial programming, you'll frequently need to transform data: convert units, apply scaling factors, normalize coordinates, or perform calculations across entire datasets. NumPy makes these operations simple and efficient.

```
# Element-wise addition - add the same value to every element
# Example: adjusting elevation readings by adding a reference height
elevation_readings = np.array([245, 312, 156, 423, 678])
sea_level_adjustment = elevation_readings + 10 # Add 10 meters to all readings
print(f"Original elevations: {elevation_readings}")
print(f"Adjusted elevations: {sea_level_adjustment}")
```

The above example shows how to perform vectorized operations on NumPy arrays. Vectorized operations are operations that are performed on entire arrays at once, rather than looping through individual elements. This is much faster than looping through individual elements.

In addition to plus and minus operations, you can also perform other mathematical operations on NumPy arrays. For example, you can perform multiplication, division, and exponentiation on NumPy arrays.

```
# Element-wise multiplication - multiply every element by the same value
# Example: converting elevation from meters to feet (1 meter = 3.28084 feet)
elevations_meters = np.array([245, 312, 156, 423, 678])
elevations_feet = elevations_meters * 3.28084
print(f"Elevations in meters: {elevations_meters}")
print(f"Elevations in feet: {elevations_feet.round(1)}") # Round to 1 decimal place

# You can also add, subtract, and divide arrays by scalars
temperatures_celsius = np.array([15.5, 22.3, 8.9, 31.2])
temperatures_fahrenheit = temperatures_celsius * 9 / 5 + 32
print(f"Temperatures in Celsius: {temperatures_celsius}")
print(f"Temperatures in Fahrenheit: {temperatures_fahrenheit.round(1)}")
```

Vectorized operations can also be performed between NumPy arrays. For example, you can perform multiplication, division, and exponentiation on NumPy arrays:

```
# Operations between arrays - element-wise operations between two arrays
# Example: calculating weighted coordinates
coordinates = np.array(
    [[35.6895, 139.6917], [40.7128, -74.0060], [51.5074, -0.1278]] # Tokyo #
    # New York
) # London

# Apply different weights to latitude and longitude (useful for some projections)
weights = np.array([1.0, 0.8]) # Weight latitude normally, reduce longitude weight
weighted_coords = coordinates * weights
print(f"Original coordinates:\n{coordinates}")
print(f"Weighted coordinates:\n{weighted_coords}")
```

```
# You can also perform operations between arrays of the same shape
coord_differences = coordinates - coordinates[0] # Distance from first city
(Tokyo)
print(f"Coordinate differences from Tokyo:\n{coord_differences}")
```

14.3.3. Reshaping Arrays

Sometimes your data comes in one format but you need it in another. For example, you might receive a flat list of coordinates that you need to organize into pairs, or you might have a 2D grid that you need to flatten for processing. NumPy's reshaping capabilities let you reorganize your data without copying it, making these transformations both fast and memory-efficient.

Important: Reshaping doesn't change the data itself, just how it's organized. The total number of elements must remain the same.

Assume you have a list of coordinates that you need to organize into pairs. You can use the `reshape` method to do this:

```
# Example: Reshape coordinate data received as a flat list
# Imagine you received GPS data as: [lat1, lon1, lat2, lon2, lat3, lon3]
flat_coordinates = np.array([35.6895, 139.6917, 40.7128, -74.0060, 51.5074,
-0.1278])
print(f"Flat coordinate data: {flat_coordinates}")

# Reshape into coordinate pairs (3 rows, 2 columns)
coordinate_pairs = flat_coordinates.reshape((3, 2))
print(f"Reshaped into coordinate pairs:\n{coordinate_pairs}")

# Alternatively, you can use -1 to let NumPy calculate one dimension
# This is useful when you know you want pairs but don't know how many
coordinate_pairs_auto = flat_coordinates.reshape(-1, 2)
print(f"Auto-calculated rows: {coordinate_pairs_auto.shape}")
```

Conversely, you can flatten a multidimensional array back to 1D. This is useful when you need to perform operations on the entire array at once. For example, you can use the `flatten` method to flatten a 2D array into a 1D array:

```
# Understanding array properties
print(f"Original flat array shape: {flat_coordinates.shape}")
print(f"Reshaped array shape: {coordinate_pairs.shape}")
print(
    f"Total elements (should be the same): {flat_coordinates.size} vs
{coordinate_pairs.size}"
)

# You can also flatten a multidimensional array back to 1D
```



```
flattened_again = coordinate_pairs.flatten()
print(f"Flattened back to 1D: {flattened_again}")
```

14.3.4. Mathematical Functions on Arrays

NumPy provides a comprehensive library of mathematical functions that operate element-wise on arrays. These functions are essential for geospatial calculations: trigonometric functions for coordinate transformations, logarithms for data normalization, and power functions for distance calculations.

The key advantage is that these functions work on entire arrays at once, making complex mathematical operations simple and fast.

```
# Mathematical functions applied to arrays
distances_squared = np.array([100, 400, 900, 1600, 2500]) # Squared distances in km²
actual_distances = np.sqrt(distances_squared) # Calculate actual distances
print(f"Squared distances: {distances_squared}")
print(f"Actual distances: {actual_distances}")

# Trigonometric functions - essential for coordinate conversions
angles_degrees = np.array([0, 30, 45, 60, 90])
angles_radians = np.radians(angles_degrees) # Convert to radians first
sine_values = np.sin(angles_radians)
cosine_values = np.cos(angles_radians)
print(f"Angles (degrees): {angles_degrees}")
print(f"Sine values: {sine_values.round(3)}")
print(f"Cosine values: {cosine_values.round(3)}")
```

```
# Logarithmic and exponential functions
population_data = np.array([100000, 500000, 1000000, 5000000, 10000000])
# Log transformation is often used to normalize highly skewed data like population
log_population = np.log10(population_data) # Base-10 logarithm
print(f"Population data: {population_data}")
print(f"Log10 of population: {log_population.round(2)}")

# Exponential functions
growth_rates = np.array([0.01, 0.02, 0.03, 0.05]) # 1%, 2%, 3%, 5% growth rates
exponential_growth = np.exp(growth_rates) # e^x
print(f"Growth rates: {growth_rates}")
print(f"Exponential factors: {exponential_growth.round(3)}")
```

14.3.5. Statistical Operations

Statistical analysis is fundamental to understanding geospatial data. NumPy provides efficient functions to calculate descriptive statistics that help you understand your data's characteristics: central tendencies (mean, median), variability (standard deviation, variance), and distributions (min, max, percentiles).

These statistics are particularly important in geospatial work for data quality assessment, outlier detection, and understanding spatial patterns.

Let's create a NumPy array of elevation readings from a hiking trail and calculate some basic statistics:

```
elevation_profile = np.array(
    [1245, 1367, 1423, 1389, 1456, 1502, 1478, 1398, 1334, 1278]
)

# Basic descriptive statistics
mean_elevation = np.mean(elevation_profile)
median_elevation = np.median(elevation_profile)
std_elevation = np.std(elevation_profile)
min_elevation = np.min(elevation_profile)
max_elevation = np.max(elevation_profile)

print(f"Elevation Profile Statistics:")
print(f"Mean elevation: {mean_elevation:.1f} meters")
print(f"Median elevation: {median_elevation:.1f} meters")
print(f"Standard deviation: {std_elevation:.1f} meters")
print(f"Elevation range: {min_elevation} - {max_elevation} meters")
print(f"Total elevation gain: {max_elevation - min_elevation} meters")

# Percentiles help understand the distribution
percentiles = np.percentile(elevation_profile, [25, 50, 75])
print(f"25th, 50th, 75th percentiles: {percentiles}")
```

14.3.6. Random Data Generation for Simulation

Random data generation is crucial for testing your geospatial algorithms, creating sample datasets, and running simulations. In geospatial work, you might need to generate random coordinates for testing, simulate sensor readings with noise, or create synthetic datasets for algorithm development.

NumPy's random module provides various functions to generate different types of random data, from uniform distributions to normal distributions and specialized patterns.

For example, you can generate random coordinates with a uniform distribution:

```
# Set a seed for reproducible results (useful for testing)
np.random.seed(42)

# Generate random coordinates within realistic bounds
# Latitudes: -90 to 90 degrees, Longitudes: -180 to 180 degrees
n_points = 5
```

```

random_latitudes = np.random.uniform(-90, 90, n_points)
random_longitudes = np.random.uniform(-180, 180, n_points)

print(f"Random coordinates:")
for i in range(n_points):
    print(f"Point {i+1}: ({random_latitudes[i]:.4f},
{random_longitudes[i]:.4f})")

```

Similarly, you can generate random temperature readings with a normal (Gaussian) distribution, which is useful for simulating measurement errors or natural variations.

```

temperature_base = 20 # Base temperature in Celsius
temperature_readings = np.random.normal(
    temperature_base, 3, 10
) # Mean=20, std=3, 10 readings
print(f"\nSimulated temperature readings (°C):")
print(temperature_readings.round(1))

# Generate random integers - useful for sampling or indexing
random_indices = np.random.randint(0, 100, 5) # Random integers between 0 and 99
print(f"\nRandom sample indices: {random_indices}")

```

14.3.7. Indexing, Slicing, and Iterating

One of NumPy's most powerful features is its flexible system for accessing and modifying array data. Whether you need to extract a single coordinate, select a region of interest, or filter data based on conditions, NumPy's indexing and slicing capabilities make these operations intuitive and efficient.

In geospatial programming, you'll constantly need to:

- Extract specific coordinates or measurements from datasets
- Select data within certain geographic bounds
- Filter data based on attribute values (elevation > 1000m, temperature < 0°C)
- Process subsets of large datasets

Let's explore the different ways to access and manipulate array data.

14.3.7.1. Indexing in NumPy

You can access individual elements of an array using their indices. Remember that NumPy arrays are zero-indexed, meaning that the first element is at index `0`.

Below are some examples of indexing 1D Arrays in NumPy.

```

# Create a 1D array of temperatures recorded at different times
temperatures = np.array([15.2, 18.5, 22.1, 19.8, 16.3])

# Accessing the first temperature reading
first_temp = temperatures[0]

```

```

print(f"First temperature reading: {first_temp}°C")

# Accessing the last temperature reading (negative indexing)
last_temp = temperatures[-1]
print(f"Last temperature reading: {last_temp}°C")

# Accessing the middle reading
middle_temp = temperatures[2]
print(f"Middle temperature reading: {middle_temp}°C")

```

In 2D arrays, you can specify both row and column indices to access a particular element.

```

# Create a 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr_2d

```

```

array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

```

```

# Accessing the element in the first row and second column
element = arr_2d[0, 1]
print(f"Element at row 1, column 2: {element}")

# Accessing the element in the last row and last column
element_last = arr_2d[-1, -1]
print(f"Element at last row, last column: {element_last}")

```

14.3.7.2. Slicing in NumPy

Slicing allows you to access a subset of an array. You can use the `:` symbol to specify a range of indices. Here is an example of slicing a 1D array in NumPy:

```

# Create a 1D array
arr = np.array([10, 20, 30, 40, 50])

# Slice elements from index 1 to 3 (exclusive)
slice_1d = arr[1:4]
print(f"Slice from index 1 to 3: {slice_1d}")

# Slice all elements from index 2 onwards
slice_2d = arr[2:]
print(f"Slice from index 2 onwards: {slice_2d}")

```

Here is an example of slicing a 2D array in NumPy:

```
# Create a 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr_2d
```

```
# Slice the first two rows and all columns
slice_2d = arr_2d[:2, :]
print(f"Sliced 2D array (first two rows):\n{slice_2d}")

# Slice the last two rows and the first two columns
slice_2d_partial = arr_2d[1:, :2]
print(f"Sliced 2D array (last two rows, first two columns):\n{slice_2d_partial}")
```

14.3.7.3. Boolean Indexing

You can also use Boolean conditions to filter elements of an array:

```
# Create a 1D array
arr = np.array([10, 20, 30, 40, 50])

# Boolean condition to select elements greater than 25
condition = arr > 25
print(f"Boolean condition: {condition}")

# Use the condition to filter the array
filtered_arr = arr[condition]
print(f"Filtered array (elements > 25): {filtered_arr}")
```

14.3.7.4. Iterating Over Arrays

You can iterate over NumPy arrays to access or modify elements. For 1D arrays, you can simply loop through the elements. For multi-dimensional arrays, you may want to iterate through rows or columns.

Here is an example of iterating over a 1D array:

```
# Create a 1D array
arr = np.array([10, 20, 30, 40, 50])

# Iterating through the array
for element in arr:
    print(f"Element: {element}")
```

Here is an example of iterating over a 2D array:

```
# Create a 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
# Iterating through rows of the 2D array
print("Iterating over rows:")
for row in arr_2d:
    print(row)

# Iterating through each element of the 2D array
print("\nIterating over each element:")
for row in arr_2d:
    for element in row:
        print(element, end=" ")
```

14.3.8. Modifying Array Elements

You can also use indexing and slicing to modify elements of the array. For example, you can set the value of an element in a 1D array:

```
# Create a 1D array
arr = np.array([10, 20, 30, 40, 50])

# Modify the element at index 1
arr[1] = 25
print(f"Modified array: {arr}")

# Modify multiple elements using slicing
arr[2:4] = [35, 45]
print(f"Modified array with slicing: {arr}")
```

```
Modified array: [10 25 30 40 50]
Modified array with slicing: [10 25 35 45 50]
```

14.3.9. Working with Geospatial Coordinates

You can use NumPy to perform calculations on arrays of geospatial coordinates, such as converting from degrees to radians.

```
# Array of latitudes and longitudes
coords = np.array([[35.6895, 139.6917], [34.0522, -118.2437], [51.5074,
-0.1278]])

# Convert degrees to radians
coords_radians = np.radians(coords)
print(f"Coordinates in radians:\n{coords_radians}")
```

14.4. Introduction to Pandas

While NumPy excels at numerical computations on arrays, real-world geospatial data often comes in more complex formats. You might have a dataset with city names, coordinates, population numbers, and dates all mixed together. This is where Pandas shines.

Pandas is built on top of NumPy but adds powerful tools for working with **structured data**—data that has labels, different data types, and relationships between columns. Think of Pandas as providing you with a powerful, programmable spreadsheet that can handle much larger datasets and more complex operations than traditional spreadsheet software.

Key Pandas concepts:

- **Series:** A one-dimensional labeled array (like a single column in a spreadsheet)
- **DataFrame:** A two-dimensional labeled data structure (like a complete spreadsheet with rows and columns)
- **Index:** Labels for rows and columns that make data selection intuitive
- **Data types:** Automatic handling of numbers, text, dates, and other data types

In geospatial programming, you'll use Pandas to:

- Load datasets from CSV files, databases, or web APIs
- Clean and prepare messy real-world data
- Filter and select subsets of your data (e.g., all cities in a specific country)
- Group and summarize data (e.g., average population by continent)
- Merge different datasets together (e.g., combining city coordinates with demographic data)

14.4.1. Creating Pandas Series and DataFrames

Let's start by creating the fundamental Pandas data structures. Understanding Series and DataFrames is crucial because they form the foundation of all data manipulation in Pandas.

A **Series** is like a single column of data with labels, while a **DataFrame** is like a complete table with multiple columns (where each column is a Series).

To use Pandas, you need to import it first. The convention is to import it as `pd`, which is short for "Pandas".

```
import pandas as pd
```

Then, you can create a Pandas Series from a list of city names:

```
# Creating a Series - a labeled one-dimensional array
# This could represent a list of city names with automatic indexing
city_series = pd.Series(["Tokyo", "Los Angeles", "London"], name="City")
print(f"Pandas Series (city names):\n{city_series}")
print(f"Series name: {city_series.name}")
print(f"Series index: {city_series.index.tolist()}")
```

Note that we assigned a name to the Series object. This is useful when you want to refer to the Series object by name later in the code.

In addition to the `Series` object, Pandas also provides a `DataFrame` object, which is a two-dimensional labeled data structure. A `DataFrame` is like a complete table with multiple columns (where each column is a `Series`). Let's create a `DataFrame` from a dictionary of data representing city information:

```
data = {
    "City": ["Tokyo", "Los Angeles", "London"],
    "Latitude": [35.6895, 34.0522, 51.5074],
    "Longitude": [139.6917, -118.2437, -0.1278],
    "Country": ["Japan", "USA", "UK"],
}
df = pd.DataFrame(data)
print(f"Pandas DataFrame (city information):\n{df}")
print(f"\nDataFrame shape: {df.shape}") # (rows, columns)
print(f"Column names: {df.columns.tolist()}")
print(f"Data types:\n{df.dtypes}")
```

Although we can use the `print` function to print the contents of a Pandas `DataFrame`, it is not formatted well. The recommended way to print a `DataFrame` is to use the `display` function or simply type the `DataFrame` name in a code cell and run it:

```
display(df)
```

The output format of the `DataFrame` object is a table with rows and columns. It is similar to a spreadsheet and looks better than the output from a `print` statement.

14.4.2. Basic DataFrame Operations

Once you have data in a `DataFrame`, you'll want to explore, filter, and transform it. Pandas provides intuitive methods for these common operations. These skills are essential because real-world geospatial datasets are rarely in exactly the format you need for analysis.

Let's explore the fundamental operations you'll use constantly in geospatial data analysis. To select a column from a `DataFrame`, you can use the square brackets notation:

```
# Selecting a specific column - returns a Series
latitudes = df["Latitude"]
print(f"Latitudes column (Series):\n{latitudes}")
print(f"Type: {type(latitudes)}")
```

You can also select multiple columns at once. This returns a new `DataFrame` with the selected columns:

```
# Selecting multiple columns - returns a DataFrame
coordinates = df[["Latitude", "Longitude"]]
print(f"\nCoordinates (DataFrame):\n{coordinates}")
print(f"Type: {type(coordinates)}")
```

You can also filter data based on conditions. For example, you can select all cities in the Western Hemisphere (negative longitude):


```
western_cities = df[df["Longitude"] < 0]
print("Cities in Western Hemisphere:")
print(western_cities)
```

You can combine multiple conditions to filter data. For example, you can select all cities with latitude greater than 40 and in the Western Hemisphere:

```
northern_western = df[(df["Latitude"] > 40) & (df["Longitude"] < 0)]
print(f"\nNorthern Western cities:\n{northern_western}")

# Filter by text values
usa_cities = df[df["Country"] == "USA"]
print(f"\nUSA cities:\n{usa_cities}")
```

Sometimes, you may want to create a new column in a DataFrame based on the values of other columns. For example, you want to convert coordinates from degrees to radians. You can do this by using the `np.radians` function applied to the columns of the DataFrame:

```
df["Lat_Radians"] = np.radians(df["Latitude"])
df["Lon_Radians"] = np.radians(df["Longitude"])
df
```

To concatenate two columns in a DataFrame, you can use the `+` operator. This is a convenient way to create a new column that combines the values of two existing columns. For example, you can create a new column that combines the city and country names:

```
df["City_Country"] = df["City"] + ", " + df["Country"]
df
```

14.4.3. Grouping and Aggregation

Pandas allows you to group data and perform aggregate functions, which is useful in summarizing large datasets. Let's create a sample DataFrame representing a list of cities and their populations:

```
data = {
    "City": ["Tokyo", "Los Angeles", "London", "Paris", "Chicago"],
    "Country": ["Japan", "USA", "UK", "France", "USA"],
    "Population": [37400068, 3970000, 9126366, 2140526, 2665000],
}
df = pd.DataFrame(data)
df
```

To group data by a column, you can use the `groupby` method. For example, you can group the data by country and calculate the total population for each country:

```
df_grouped = df.groupby("Country")["Population"].sum()
print(f"Total Population by Country:\n{df_grouped}")
```

14.4.4. Merging DataFrames

Merging datasets is essential when combining different geospatial datasets, such as joining city data with demographic information. Assuming you have two DataFrames, one with city data and one with demographic information, you can merge them using the `merge` function with the `on` parameter:

```
df1 = pd.DataFrame(
    {"City": ["Tokyo", "Los Angeles", "London"], "Country": ["Japan", "USA",
    "UK"]}
)
df2 = pd.DataFrame(
    {
        "City": ["Tokyo", "Los Angeles", "London"],
        "Population": [37400068, 3970000, 9126366],
    }
)
```

df1

df2

```
# Merge the two DataFrames on the 'City' column
df_merged = pd.merge(df1, df2, on="City")
df_merged
```

14.4.5. Handling Missing Data

In real-world datasets, missing data is common. Pandas provides tools to handle missing data, such as filling or removing missing values.

```
data_with_nan = {
    "City": ["Tokyo", "Los Angeles", "London", "Paris"],
    "Population": [37400068, 3970000, None, 2140526],
}
df_nan = pd.DataFrame(data_with_nan)
df_nan
```

You can fill missing values with the mean of the column. For example, you can fill the missing population with the mean population of the cities:

```
df_filled = df_nan.fillna(df_nan["Population"].mean())
df_filled
```

14.4.6. Reading Geospatial Data from a CSV File

Pandas can read and write data in various formats, such as CSV, Excel, and SQL databases. This makes it easy to load and save data from different sources. For example, you can read a CSV file into a Pandas DataFrame and then perform operations on the data.

Let's read a CSV file from an HTTP URL into a Pandas DataFrame:

```
url = "https://github.com/opengeos/datasets/releases/download/world/world_cities.csv"
df = pd.read_csv(url)
df.head()
```

The DataFrame contains information about world cities, including their names, countries, populations, and geographical coordinates. We can calculate the total population of all cities in the dataset using NumPy and Pandas as follows.

```
np.sum(df["population"])
```

14.4.7. Creating plots with Pandas

Pandas provides built-in plotting capabilities that allow you to create various types of plots directly from DataFrames.

```
# Load the dataset from an online source
url = "https://raw.githubusercontent.com/pandas-dev/pandas/main/doc/data/air_quality_no2.csv"
air_quality = pd.read_csv(url, index_col=0, parse_dates=True)

# Display the first few rows of the dataset
air_quality.head()
```

To do a quick visual check of the data, you can use the `plot` method of the `DataFrame` object. This will create a line plot of the data (Figure 9).

```
import matplotlib.pyplot as plt

# Set default DPI for all plots
plt.rcParams['figure.dpi'] = 300
```

```
air_quality.plot(figsize=(10, 5))
```

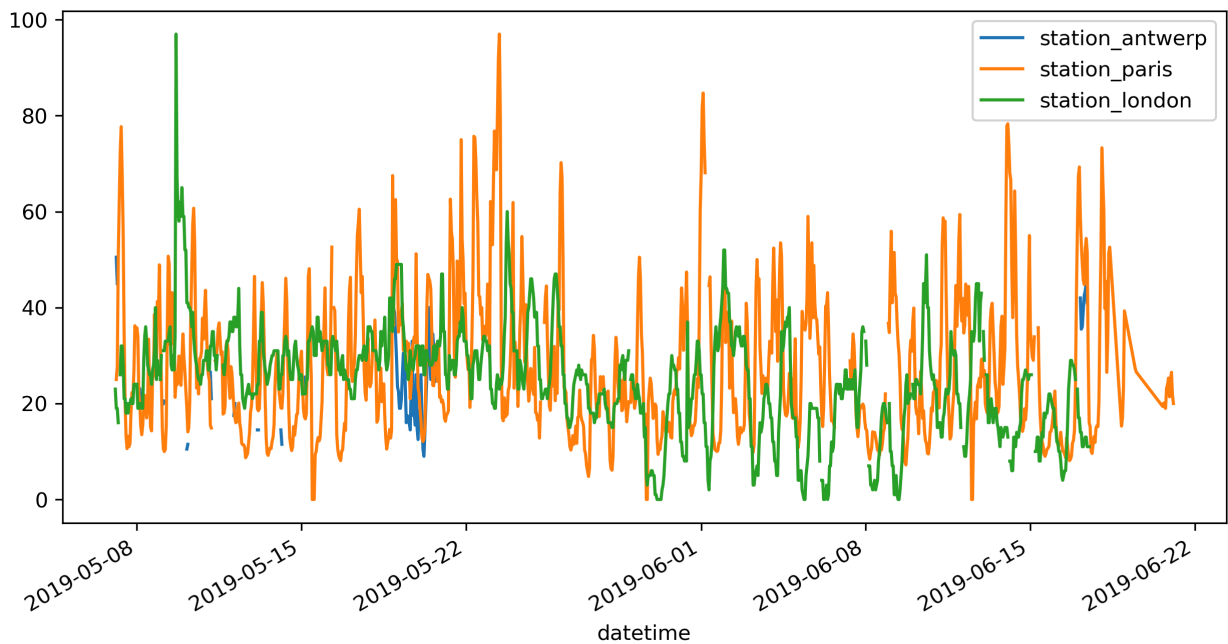


Figure 9: A line plot of air quality data in three cities.

To plot only the columns of the data table with the data from Paris ([Figure 10](#)):

```
air_quality["station_paris"].plot(figsize=(10, 5))
```

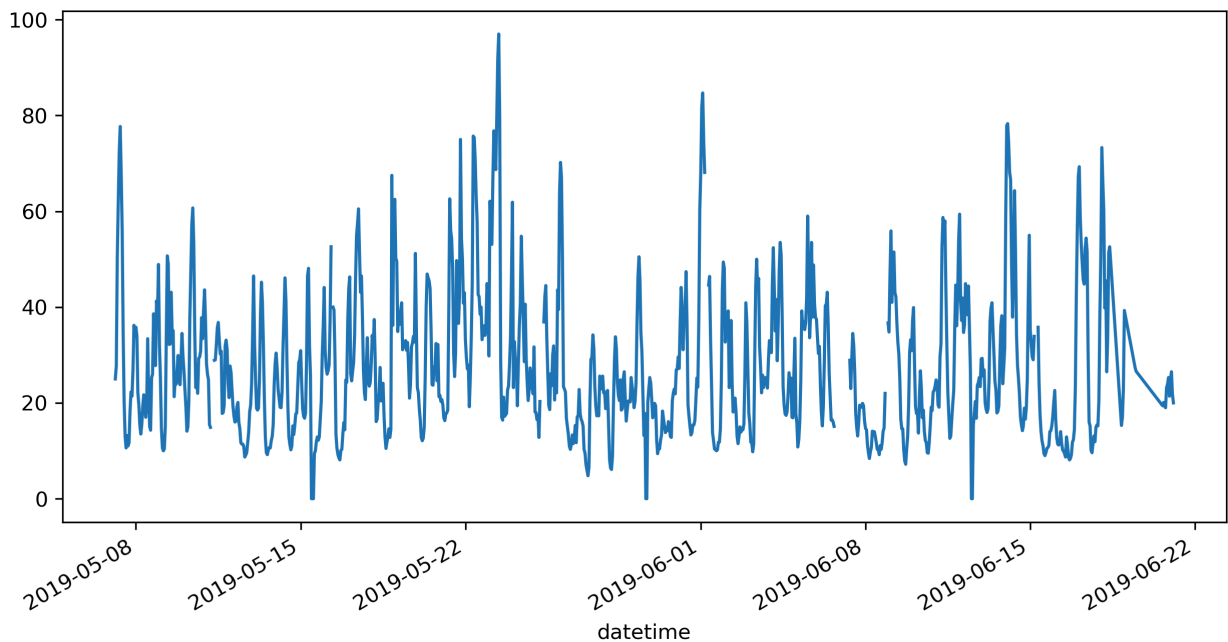


Figure 10: A line plot of air quality data in Paris.

To visually compare the values measured in London versus Paris, you can use a scatter plot (see [Figure 11](#)):

```
air_quality.plot.scatter(x="station_london", y="station_paris", alpha=0.5)
```

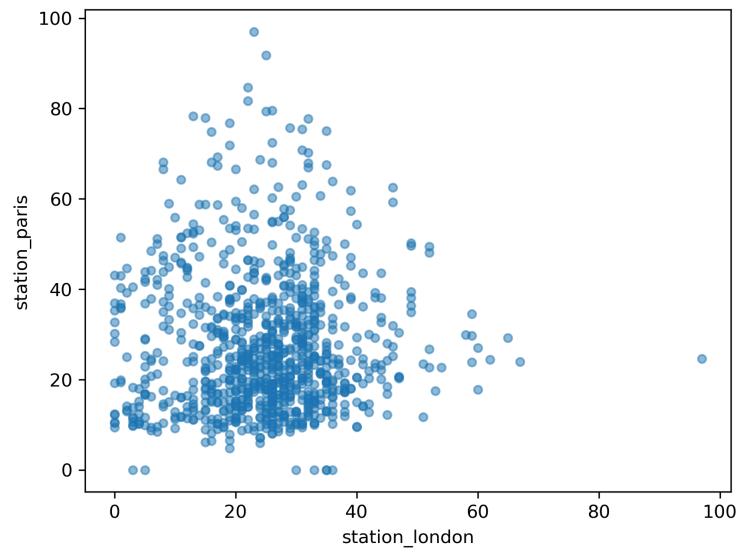


Figure 11: A scatter plot of air quality data in Paris and London.

To visualize each of the columns in a separate subplot, you can use the `plot.area` method. This will create an area plot of the data (Figure 12):

```
air_quality.plot.area(figsize=(12, 4), subplots=True)
```

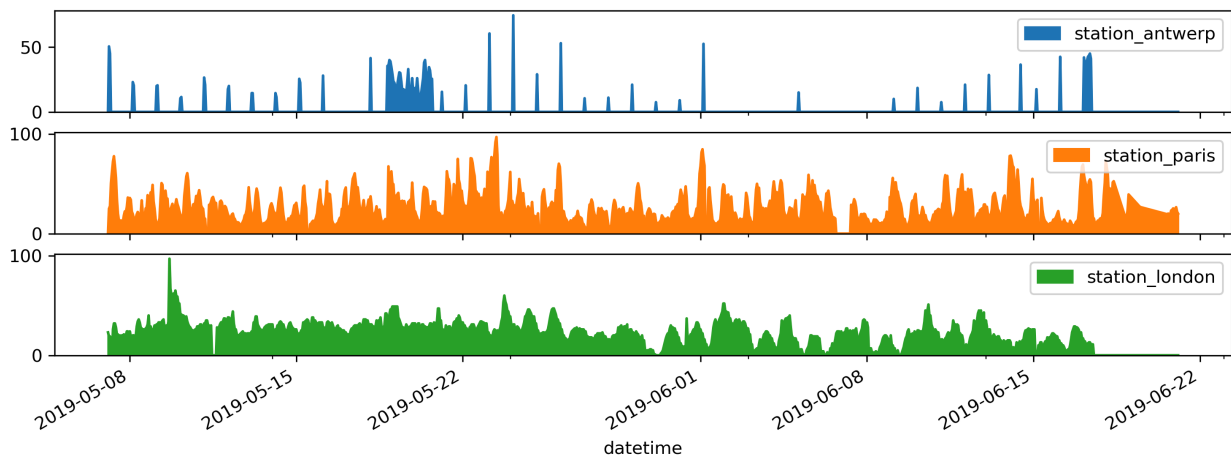


Figure 12: An area plot of air quality data in three cities.

14.5. Combining NumPy and Pandas

You can combine NumPy and Pandas to perform complex data manipulations. For instance, you might want to apply NumPy functions to a Pandas DataFrame or use Pandas to organize and visualize the results of NumPy operations.

Let's say you have a dataset of cities, and you want to calculate the average distance from each city to all other cities.

```
# Define the Haversine formula using NumPy
def haversine_np(lat1, lon1, lat2, lon2):
    R = 6371.0 # Earth radius in kilometers
    dlat = np.radians(lat2 - lat1)
    dlon = np.radians(lon2 - lon1)
    a = (
        np.sin(dlat / 2) ** 2
        + np.cos(np.radians(lat1)) * np.cos(np.radians(lat2)) * np.sin(dlon / 2)
    ** 2
    )
    c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a))
    distance = R * c
    return distance

# Define a function to calculate distances from a city to all other cities
def calculate_average_distance(df):
    lat1 = df["Latitude"].values
    lon1 = df["Longitude"].values
    lat2, lon2 = np.meshgrid(lat1, lon1)
    distances = haversine_np(lat1, lon1, lat2, lon2)
    avg_distances = np.mean(distances, axis=1)
    return avg_distances

# Creating a DataFrame
data = {
    "City": ["Tokyo", "Los Angeles", "London"],
    "Latitude": [35.6895, 34.0522, 51.5074],
    "Longitude": [139.6917, -118.2437, -0.1278],
}
df = pd.DataFrame(data)

# Apply the function to calculate average distances
df["Avg_Distance_km"] = calculate_average_distance(df)
df
```

14.6. Key Takeaways

NumPy and Pandas form the data analysis foundation for virtually all geospatial programming in Python. Understanding these libraries will dramatically improve your ability to work with real-world geospatial datasets efficiently and effectively.

NumPy Fundamentals:

- **Arrays are faster and more memory-efficient** than Python lists for numerical data

- **Vectorization** allows you to perform operations on entire arrays without writing loops
- **Broadcasting** enables operations between arrays of different shapes
- **Mathematical functions** work element-wise on arrays, making complex calculations simple
- **Array reshaping** helps you reorganize data without copying it

Pandas Fundamentals:

- **DataFrames** provide a powerful structure for mixed-type tabular data
- **Indexing and selection** make it easy to extract subsets of your data
- **Filtering** allows you to select data based on conditions (geographic bounds, attribute values)
- **Column operations** enable you to create new calculated fields
- **Data integration** tools help you combine datasets from different sources

Why This Matters for Geospatial Work:

- Most geospatial Python libraries (GeoPandas, Rasterio, etc.) are built on NumPy and Pandas
- Real-world geospatial data is often messy and requires cleaning and transformation
- Efficient data processing becomes critical when working with large datasets
- These skills transfer directly to more advanced geospatial analysis techniques

14.7. Further Reading

To learn more about NumPy and Pandas, you can refer to the following resources:

- [NumPy Documentation](https://numpy.org/doc/stable)⁴⁰
- [Pandas Documentation](https://pandas.pydata.org/docs)⁴¹

Reading package documentation is a great way to learn more about the functions and methods available in NumPy and Pandas. In this chapter, we have only touched on the surface of what these libraries can do. You will find more examples and exercises in the exercises section below.

14.8. Exercises

14.8.1. Exercise 1: NumPy Array Operations and Geospatial Coordinates

Practice fundamental NumPy operations with geospatial coordinate data.

Tasks:

1. **Create coordinate arrays:** Build a 2D NumPy array containing the latitude and longitude of these cities:
 - Tokyo: (35.6895, 139.6917)
 - New York: (40.7128, -74.0060)
 - London: (51.5074, -0.1278)
 - Paris: (48.8566, 2.3522)
2. **Unit conversion:** Convert all coordinates from degrees to radians using `np.radians()`
3. **Distance calculations:** Calculate the coordinate differences between Tokyo and each other city (in radians)

⁴⁰<https://numpy.org/doc/stable>

⁴¹<https://pandas.pydata.org/docs>

4. **Statistical analysis:** Find the mean latitude and longitude, and identify which city is furthest from the mean coordinates

14.8.2. Exercise 2: Pandas DataFrame Operations with Geospatial Data

Apply Pandas skills to analyze a real-world geospatial dataset with city population data.

Tasks:

1. **Data loading:** Load the world cities dataset from: https://github.com/opengeos/datasets/releases/download/world/world_cities.csv
2. **Data exploration:**
 - Display the first 5 rows to understand the data structure
 - Check the shape of the dataset (how many cities and columns?)
 - Identify any missing values using `df.isnull().sum()`
3. **Data filtering:**
 - Filter to include only cities with population > 1 million
 - Create a subset of cities in a specific continent or region of your choice
4. **Data aggregation:**
 - Group cities by country and calculate total population per country
 - Find the country with the most cities in the dataset
 - Calculate average population by continent (if continent data is available)
5. **Data analysis:**
 - Sort cities by population and display the top 10 largest cities
 - Find the northernmost and southernmost cities in the dataset
 - Calculate the population density if area data is available

Section III: Geospatial Programming with Python

Chapter 15. Introduction to Geospatial Python

15.1. Introduction

Python has emerged as the dominant language for geospatial analysis and visualization, offering a rich ecosystem of libraries that handle everything from basic data manipulation to advanced cloud computing and 3D mapping. This chapter introduces you to the comprehensive suite of geospatial tools covered in this book, helping you understand how different libraries work together to create powerful analytical workflows.

The geospatial Python ecosystem is built on a foundation of interoperable libraries, each serving specific purposes while working seamlessly together. Understanding the relationships between these tools—and knowing when to use each one—is crucial for building effective geospatial applications.

15.2. The Geospatial Python Ecosystem

The libraries covered in this book can be organized into several functional categories:

15.2.1. Foundation Libraries

GDAL/OGR forms the bedrock of geospatial computing in Python. This C++ library provides universal format support for raster and vector data, coordinate transformations, and data conversion capabilities. Most other geospatial libraries build upon GDAL's capabilities.

15.2.2. Data Structures and Analysis

GeoPandas extends pandas to handle vector data (points, lines, polygons), providing spatial operations, coordinate reference system management, and integration with the broader pandas ecosystem.

Xarray handles multi-dimensional labeled datasets, making it ideal for climate data, satellite imagery time series, and any data that varies across multiple dimensions like time, latitude, and longitude.

Rasterio provides a Pythonic interface to GDAL for raster data, offering memory-efficient reading and writing of satellite imagery, digital elevation models, and gridded datasets.

Rioxarray bridges Xarray and Rasterio, adding geospatial capabilities to Xarray datasets including coordinate reference systems, spatial indexing, and raster operations.

15.2.3. Interactive Visualization

Leafmap offers a unified interface for creating interactive maps using multiple backends (folium, ipyleaflet, maplibre). It simplifies map creation, data visualization, and provides specialized tools for geospatial analysis.

MapLibre enables advanced 2D and 3D mapping with modern web technologies, supporting vector tiles, globe projections, and sophisticated styling capabilities.

HyperCoast supports 3D visualization of hyperspectral data with a few lines of code.

15.2.4. Specialized Analysis

WhiteboxTools provides over 550 geoprocessing tools with a focus on hydrological analysis, terrain processing, and LiDAR data analysis, all optimized for performance.

Geemap connects Python to Google Earth Engine’s planetary-scale computing platform, enabling analysis of petabytes of satellite imagery and geospatial datasets in the cloud.

DuckDB offers fast analytical database capabilities with spatial extensions, providing SQL-based analysis of geospatial data with excellent performance characteristics.

Apache Sedona provides distributed computing capabilities for geospatial data processing, enabling processing of large datasets across multiple nodes.

15.2.5. Application Development

Voila and Solara enable the creation of interactive web applications from Python code and Jupyter notebooks, allowing you to deploy geospatial analyses as user-friendly web interfaces.

15.3. Understanding Library Relationships

The power of the geospatial Python ecosystem comes from how these libraries complement each other:

- **GDAL** provides the foundation that **Rasterio**, **GeoPandas**, and **Rioxarray** build upon
- **Xarray** and **Rioxarray** work together for multi-dimensional geospatial data
- **GeoPandas** handles vector data that can be visualized with **Leafmap** or **MapLibre**
- **Leafmap** serves as a high-level interface that can utilize **MapLibre**, **Folium**, or **Ipyleaflet** backends
- **WhiteboxTools** performs analysis that can be visualized with **Leafmap**
- **Geemap** brings cloud computing capabilities to **Leafmap**’s visualization framework
- **DuckDB** provides database capabilities that complement file-based workflows
- **Solara** deploys any of these tools as web applications

15.4. Setting Up Your Environment

The geospatial Python ecosystem requires careful dependency management due to the complexity of underlying C/C++ libraries. We recommend several approaches depending on your needs and experience level.

15.4.1. Option 1: Using uv (Recommended for Beginners)

uv⁴² is an extremely fast Python package manager that simplifies geospatial library installation:

```
# Install uv
# macOS and Linux:
curl -Lsf https://astral.sh/uv/install.sh | sh

# Windows:
powershell -ExecutionPolicy Bypass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

⁴²<https://github.com/astral-sh/uv>

```
# Create virtual environment
uv venv

# Activate environment
# macOS and Linux:
source .venv/bin/activate

# Windows:
.venv\Scripts\activate

# Install geospatial package (includes most libraries covered in this book)
uv pip install --find-links https://girder.github.io/large_image_wheels gdal
pyproj
uv pip install pygis
```

15.4.2. Option 2: Using pixi (For Complex Dependencies)

[Pixi](https://pixi.sh)⁴³ excels at managing complex geospatial dependencies from conda-forge:

```
# Install pixi
# macOS and Linux:
curl -fsSL https://pixi.sh/install.sh | bash

# Windows:
iwr -useb https://pixi.sh/install.ps1 | iex

# Initialize project and add dependencies
pixi init
pixi add pygis jupyterlab
pixi run jupyter lab
```

15.4.3. Option 3: Using conda/mamba (Traditional Approach)

```
# Create environment
conda create -n geo python=3.12
conda activate geo

# Install from conda-forge
conda install -c conda-forge mamba
mamba install -c conda-forge pygis
```

⁴³<https://pixi.sh>

15.5. Verification and First Steps

Once your environment is set up, verify that key libraries are working correctly:

```
# Test core geospatial libraries
import geopandas as gpd
import rasterio
import xarray as xr
import rioxarray
import leafmap
import pandas as pd
import numpy as np

print("✓ All core libraries imported successfully!")
```

15.5.1. Create Your First Interactive Map

Let's create a simple interactive map to verify everything is working:

```
# Create an interactive map using Leafmap
m = leafmap.Map(center=[40, -100], zoom=4, height="500px")

# Add different basemap options
m.add_basemap("OpenTopoMap")
m.add_basemap("USGS.Imagery")

# Display the map
m
```

If you see an interactive map with multiple basemap options, your geospatial Python environment is ready for the more advanced topics covered in subsequent chapters.

15.6. Learning Path and Chapter Overview

This book is structured to build your geospatial Python skills progressively. Here's the recommended learning path:

15.6.1. Foundation (Start Here)

1. **Vector Data Analysis with GeoPandas** - Learn to work with points, lines, and polygons
2. **Working with Raster Data Using Rasterio** - Handle satellite imagery and gridded datasets
3. **Multi-dimensional Data Analysis with Xarray** - Work with climate data and time series
4. **Raster Analysis with Rioxarray** - Combine Xarray's power with geospatial capabilities

15.6.2. Visualization and Interaction

5. **Interactive Visualization with Leafmap** - Create interactive maps and visualizations
6. **3D Mapping with MapLibre** - Build advanced 2D and 3D visualizations

7. 3D Visualization of Hyperspectral Data with HyperCoast - Visualize hyperspectral data in 3D

15.6.3. Specialized Analysis

8. **Geoprocessing with WhiteboxTools** - Perform advanced spatial analysis and hydrological modeling
9. **Cloud Computing with Earth Engine and Geemap** - Access planetary-scale datasets through Google Earth Engine
10. **Spatial Database Analysis with DuckDB** - Use SQL for spatial data analysis
11. **Distributed Computing with Apache Sedona** - Process large geospatial datasets in a distributed environment

15.6.4. Foundation and Integration

12. **Geospatial Data Processing with GDAL and OGR** - Understand the foundation libraries and data format handling

15.6.5. Application Development

13. **Building Interactive Dashboards with Voilà and Solara** - Deploy your analyses as web applications

15.7. Key Concepts to Remember

As you work through this book, keep these fundamental concepts in mind:

Data Types: Geospatial data comes in two main types—vector data (discrete features like points, lines, polygons) and raster data (continuous grids like satellite images or elevation models).

Coordinate Reference Systems (CRS): All geospatial data must be referenced to a coordinate system. Understanding and managing CRS is crucial for accurate analysis.

Scale and Resolution: Different datasets have different spatial and temporal scales. Matching these appropriately is essential for meaningful analysis.

Memory Management: Geospatial datasets can be very large. Learning to work with data efficiently—reading only what you need, using appropriate data types, and leveraging lazy evaluation—is crucial.

Format Diversity: The geospatial world uses many data formats. Understanding when to use GeoJSON vs. Shapefile vs. GeoParquet vs. GeoTIFF vs. NetCDF will improve your workflows.

Integration: The real power comes from combining multiple libraries. A typical workflow might use GeoPandas for vector analysis, Rasterio for raster processing, Leafmap for visualization, and Solara for deployment.

15.8. Getting Help and Resources

The geospatial Python community is active and supportive. Key resources include:

- **Documentation:** Each library has comprehensive documentation with examples
- **GitHub Issues:** Report bugs and request features on each library's GitHub repository
- **Stack Overflow:** Search for existing questions or ask new ones with appropriate tags
- **Community Forums:** Join discussions on platforms like GIS Stack Exchange

- **Conferences:** Attend SciPy, FOSS4G, or CNG conferences for networking and learning

15.9. Next Steps

Now that you understand the geospatial Python ecosystem and have verified your installation, you're ready to dive into specific libraries. We recommend starting with GeoPandas if you're primarily working with vector data, or Rasterio if you're focused on satellite imagery and raster analysis.

Each subsequent chapter builds on concepts introduced here while diving deep into specific capabilities. The hands-on examples and exercises in each chapter will help you develop practical skills for real-world geospatial analysis and visualization projects.

Remember that mastering geospatial Python is a journey—start with the basics, practice regularly, and gradually incorporate more advanced techniques as your confidence and needs grow. The comprehensive toolkit you'll learn in this book will serve you well across a wide range of geospatial applications, from environmental monitoring to urban planning to scientific research.

15.10. Exercises

1. **Environment Setup:** Complete the installation process for your preferred method (uv, pixi, or conda) and verify that all core libraries import correctly.
2. **First Map:** Create an interactive map centered on your location or area of interest. Experiment with different basemaps and zoom levels.
3. **Library Exploration:** Choose one library from each category (data structures, visualization, specialized analysis) and read through their documentation to understand their primary use cases.
4. **Integration Planning:** Think about a geospatial project you'd like to work on. Based on the library descriptions, identify which tools you would need and how they might work together in your workflow.

These exercises will prepare you for the more detailed exploration of each library in the following chapters.

Chapter 16. Vector Data Analysis with GeoPandas

16.1. Introduction

Working with geospatial data in Python becomes remarkably intuitive with [GeoPandas](https://geopandas.org)⁴⁴, a powerful library that brings the familiar pandas experience to geographic data analysis. Just as pandas revolutionized data science by making tabular data manipulation simple and intuitive, GeoPandas does the same for spatial vector data—points, lines, and polygons that represent real-world geographic features.

Imagine you're analyzing urban development patterns, studying wildlife migration routes, or mapping disease outbreaks. These scenarios involve spatial relationships that traditional data analysis tools struggle to handle effectively. GeoPandas bridges this gap by seamlessly integrating geospatial operations with the pandas interface you already know, while leveraging the geometric capabilities of [Shapely](https://github.com/shapely/shapely)⁴⁵ under the hood.

What makes GeoPandas particularly powerful is its ability to handle both the geometric aspects of spatial data (where features are located, their shapes and sizes) and their attribute data (what those features represent—population, elevation, land use type, etc.). This dual nature allows you to perform complex spatial analyses while maintaining the familiar DataFrame operations you use in pandas.

Key capabilities of GeoPandas:

- **Unified data model:** Store geometric shapes alongside their attributes in a single data structure
- **Spatial operations:** Calculate distances, intersections, buffers, and other geometric relationships
- **Coordinate system management:** Handle different map projections and coordinate reference systems
- **Multiple format support:** Read and write industry-standard formats like Shapefile, GeoJSON, GeoPackage, and more
- **Visualization integration:** Create maps using Matplotlib with simple, pandas-style plotting methods

16.2. Learning Objectives

By the end of this chapter, you should be able to:

- Understand the basic data structures in GeoPandas: `GeoDataFrame` and `GeoSeries`
- Create `GeoDataFrames` from tabular data and geometric shapes
- Read and write geospatial data formats like Shapefile and GeoJSON
- Perform common geospatial operations such as measuring areas, distances, and spatial relationships
- Visualize geospatial data using [Matplotlib](https://matplotlib.org)⁴⁶ and GeoPandas' built-in plotting functions
- Work with different Coordinate Reference Systems (CRS) and project geospatial data

16.3. Core Concepts

Understanding GeoPandas begins with grasping its fundamental data structures, which extend pandas to handle geometric data. These structures form the foundation for all spatial analysis operations.

⁴⁴<https://geopandas.org>

⁴⁵<https://github.com/shapely/shapely>

⁴⁶<https://matplotlib.org>

16.3.1. GeoDataFrame and GeoSeries

The **GeoDataFrame** is the central data structure in GeoPandas. Think of it as a pandas DataFrame with superpowers—it contains all the familiar tabular data capabilities you expect, plus a special **geometry** column that stores spatial shapes. This geometry column can contain points (representing locations like cities or GPS coordinates), lines (representing features like rivers or roads), or polygons (representing areas like countries or land parcels).

The **GeoSeries** handles the geometric data within a GeoDataFrame. While you'll primarily work with GeoDataFrames, understanding that GeoSeries manages the actual geometric operations helps clarify how spatial methods work.

16.3.2. Active Geometry Concept

A crucial concept in GeoPandas is that while a GeoDataFrame can have multiple geometry columns (useful for storing both original and processed shapes), only one geometry column is “active” at any time. All spatial operations—calculating areas, measuring distances, creating buffers—operate on this active geometry, which you access through the **.geometry** attribute. This design provides flexibility while maintaining clarity about which shapes are being analyzed.

16.4. Installing GeoPandas

Before we begin, ensure you have GeoPandas and related dependencies installed:

```
# %pip install geopandas pygis
```

Import the necessary libraries. Notice how we import both pandas and GeoPandas—they work together seamlessly:

```
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
```

16.5. Creating GeoDataFrames

Understanding how to create GeoDataFrames from scratch provides insight into their structure and helps when you need to convert coordinate data into spatial objects.

16.5.1. Creating Points from Coordinate Data

Often, you'll start with tabular data containing latitude and longitude coordinates that need to be converted into geometric points. This is a common scenario when working with GPS data, survey points, or any location-based datasets:

```
# Creating a GeoDataFrame from coordinate data
data = {
    "City": ["Tokyo", "New York", "London", "Paris"],
```

```

    "Latitude": [35.6895, 40.7128, 51.5074, 48.8566],
    "Longitude": [139.6917, -74.0060, -0.1278, 2.3522],
}

# First create a regular pandas DataFrame
df = pd.DataFrame(data)

# Convert to GeoDataFrame by creating Point geometries from coordinates
gdf = gpd.GeoDataFrame(df, geometry=gpd.points_from_xy(df.Longitude,
df.Latitude))
gdf

```

	City	Latitude	Longitude	geometry
0	Tokyo	35.6895	139.6917	POINT (139.6917 35.6895)
1	New York	40.7128	-74.0060	POINT (-74.006 40.7128)
2	London	51.5074	-0.1278	POINT (-0.1278 51.5074)
3	Paris	48.8566	2.3522	POINT (2.3522 48.8566)

The `gpd.points_from_xy()` function is a convenience method that creates Point geometries from separate longitude (x) and latitude (y) columns. Notice that longitude comes first (x-coordinate), then latitude (y-coordinate)—this follows the standard mathematical convention, though it might feel backwards since we often think “latitude, longitude” when speaking.

16.6. Reading and Writing Geospatial Data

Real-world geospatial analysis typically begins with existing datasets stored in various formats. GeoPandas excels at reading these formats, with automatic handling of coordinate systems, attribute data, and complex geometries.

16.6.1. Understanding Geospatial File Formats

Different geospatial formats serve different purposes:

- **GeoJSON:** Web-friendly, human-readable format ideal for web mapping and data sharing
- **Shapefile:** Traditional GIS format, widely supported but consists of multiple files
- **GeoPackage:** Modern, single-file format that can store multiple layers and data types

16.6.2. Reading a GeoJSON File

Let’s load a real-world dataset—New York City borough boundaries—to demonstrate practical GeoPandas usage. This dataset contains polygon geometries representing the five NYC boroughs along with attribute data about each borough:

```

url = "https://github.com/opengeos/datasets/releases/download/vector/nybb.
geojson"

```

```
gdf = gpd.read_file(url)
gdf.head()
```

	BoroCode	BoroName	Shape_Leng	Shape_Area	geometry
0	5	Staten Island	330470.010332	1.623820e+09	MULTIPOLYGON (((970217.022 145643.332, 970227....
1	4	Queens	896344.047763	3.045213e+09	MULTIPOLYGON (((1029606.077 156073.814, 102957...
2	3	Brooklyn	741080.523166	1.937479e+09	MULTIPOLYGON (((1021176.479 151374.797, 102100...
3	1	Manhattan	359299.096471	6.364715e+08	MULTIPOLYGON (((981219.056 188655.316, 980940....
4	2	Bronx	464392.991824	1.186925e+09	MULTIPOLYGON (((1012821.806 229228.265, 101278...

The `read_file()` function automatically detects the file format and coordinate system, loading both the geometric shapes and their associated attributes. This GeoDataFrame contains several informative columns:

- **BoroCode** : Numeric codes for each borough
- **BoroName** : Human-readable borough names
- **Shape_Leng** and **Shape_Area** : Geometric measurements
- **geometry** : The actual polygon shapes defining each borough boundary

16.6.3. Writing Geospatial Data

GeoPandas makes saving processed data straightforward, supporting multiple output formats. The ability to save your work in different formats provides flexibility for sharing data with different software systems:

```
output_file = "nyc_boroughs.geojson"
gdf.to_file(output_file, driver="GeoJSON")
print(f"GeoDataFrame has been written to {output_file}")
```

The `driver` parameter explicitly specifies the output format, though GeoPandas can often infer the format from the file extension. This approach ensures your data is saved exactly as intended.

You can also export to other formats commonly used in GIS workflows:

```
# Save as Shapefile (traditional GIS format)
output_file = "nyc_boroughs.shp"
gdf.to_file(output_file)

# Save as GeoPackage (modern, single-file format)
output_file = "nyc_boroughs.gpkg"
gdf.to_file(output_file, driver="GPKG")
```

16.7. Projections and Coordinate Reference Systems (CRS)

Understanding coordinate reference systems is fundamental to geospatial analysis. All locations on Earth need to be referenced to a coordinate system, and different systems serve different purposes.

16.7.1. Understanding Coordinate Systems

Think of coordinate reference systems as different ways to represent the curved Earth on flat maps or in computer systems. Geographic coordinate systems use latitude and longitude (like GPS coordinates), while projected coordinate systems convert these to flat, Cartesian coordinates measured in linear units like meters or feet.

16.7.2. Checking and Understanding CRS

Every GeoDataFrame has a coordinate reference system that determines how coordinates are interpreted:

```
print(f"Current CRS: {gdf.crs}")
```

```
Current CRS: EPSG:3857
```

This dataset uses [EPSG:2263](https://epsg.io/2263)⁴⁷ (NAD83 / New York Long Island State Plane), a projected coordinate system designed specifically for the New York area. The measurements are in feet, which is typical for US state plane coordinate systems.

[EPSG](https://epsg.io)⁴⁸ codes are standardized identifiers for coordinate reference systems, maintained by the International Association of Oil & Gas Producers (formerly the European Petroleum Survey Group). These codes provide a universal way to specify exactly which coordinate system is being used.

16.7.3. Reprojecting to Different Coordinate Systems

Different coordinate systems are appropriate for different types of analysis. For global visualization, we often use geographic coordinates (latitude/longitude). For area calculations, we need projected coordinates with meaningful linear units:

```
# Reproject to WGS84 (latitude/longitude) for global compatibility
gdf_4326 = gdf.to_crs(epsg=4326)
print(f"Reprojected CRS: {gdf_4326.crs}")
gdf_4326.head()
```

	BoroCode	BoroName	Shape_Leng	Shape_Area	geometry
0	5	Staten Island	330470.010332	1.623820e+09	MULTIPOLYGON (((-74.05051 40.56642, -74.05047 ...
1	4	Queens	896344.047763	3.045213e+09	MULTIPOLYGON (((-73.83668 40.59495, -73.83678 ...
2	3	Brooklyn	741080.523166	1.937479e+09	MULTIPOLYGON (((-73.86706 40.58209, -73.86769 ...
3	1	Manhattan	359299.096471	6.364715e+08	MULTIPOLYGON (((-74.01093 40.68449, -74.01193 ...
4	2	Bronx	464392.991824	1.186925e+09	MULTIPOLYGON (((-73.89681 40.79581, -73.89694 ...

Notice how the coordinate values in the `geometry` column have changed from large numbers (representing feet in the state plane system) to decimal degrees (representing latitude and longitude). This

⁴⁷<https://epsg.io/2263>

⁴⁸<https://epsg.io>

transformation preserves the shapes and spatial relationships while changing how coordinates are expressed.

16.8. Spatial Measurements and Analysis

One of GeoPandas' most powerful features is its ability to perform spatial measurements and geometric analysis directly on DataFrame-like structures.

16.8.1. Preparing Data for Accurate Measurements

For accurate area and distance calculations, we need a coordinate system with meaningful linear units. Web Mercator ([EPSG:3857](https://epsg.io/3857))⁴⁹ is commonly used for web mapping and provides measurements in meters:

```
# Reproject to Web Mercator for accurate area calculations in square meters
gdf = gdf.to_crs("EPSG:3857")

# Set BoroName as index for easier data access
gdf = gdf.set_index("BoroName")
print(f"Now using CRS: {gdf.crs}")
```

16.8.2. Calculating Areas

With the data in an appropriate coordinate system, we can calculate meaningful area measurements:

```
# Calculate area in square meters
gdf["area"] = gdf.area

# Convert to more readable units (square kilometers)
gdf["area_km2"] = gdf["area"] / 1_000_000

# Display results sorted by area
gdf[["area", "area_km2"]].sort_values("area_km2", ascending=False)
```

	area	area_km2
BoroName		
Queens	4.928308e+08	492.830760
Brooklyn	3.129684e+08	312.968357
Staten Island	2.618039e+08	261.803854
Bronx	1.929250e+08	192.925032
Manhattan	1.032201e+08	103.220068

⁴⁹<https://epsg.io/3857>

These calculations reveal that Queens is the largest NYC borough by area, followed by Brooklyn and Staten Island. This type of analysis is fundamental for urban planning, resource allocation, and demographic studies.

16.8.3. Extracting Geometric Features

GeoPandas provides methods to extract different geometric representations from complex shapes:

```
# Extract boundary lines from polygons
gdf["boundary"] = gdf.boundary

# Calculate centroids (geometric centers)
gdf["centroid"] = gdf.centroid

# Display the geometric features
gdf[["boundary", "centroid"]].head()
```

	boundary	centroid
BoroName		
Staten Island	MULTILINESTRING ((-8243264.88 4948597.813, -82...	POINT (-8254713.581 4950718.033)
Queens	MULTILINESTRING ((-8219461.955 4952778.645, -8...	POINT (-8217436.761 4969318.623)
Brooklyn	MULTILINESTRING ((-8222843.701 4950893.717, -8...	POINT (-8231817.476 4960085.209)
Manhattan	MULTILINESTRING ((-8238858.852 4965914.967, -8...	POINT (-8233984.776 4979551.696)
Bronx	MULTILINESTRING ((-8226155.114 4982269.852, -8...	POINT (-8222783.625 4990631.161)

Boundaries convert polygon areas to their outline as LineString geometries, useful for analyzing borders or creating simplified representations. **Centroids** provide single-point representations of complex shapes, valuable for distance calculations and spatial indexing.

16.8.4. Distance Calculations

Spatial distance analysis helps answer questions like “How far is each borough from Manhattan?” This type of analysis is crucial for accessibility studies, service area planning, and transportation analysis:

```
# Use Manhattan's centroid as the reference point
manhattan_centroid = gdf.loc["Manhattan", "centroid"]

# Calculate distance from each borough centroid to Manhattan
gdf["distance_to_manhattan"] = gdf["centroid"].distance(manhattan_centroid)

# Convert to kilometers and display results
gdf["distance_to_manhattan_km"] = gdf["distance_to_manhattan"] / 1000

gdf[["distance_to_manhattan_km"]].sort_values("distance_to_manhattan_km")
```

	distance_to_manhattan_km
BoroName	
Manhattan	0.000000
Bronx	15.755010
Queens	19.456427
Brooklyn	19.586764
Staten Island	35.511456

16.8.5. Statistical Analysis of Spatial Data

Combining spatial measurements with statistical analysis provides deeper insights:

```
# Calculate summary statistics
mean_distance = gdf["distance_to_manhattan_km"].mean()
max_distance = gdf["distance_to_manhattan_km"].max()
total_area = gdf["area_km2"].sum()

print(f"Mean distance to Manhattan: {mean_distance:.2f} km")
print(f"Maximum distance to Manhattan: {max_distance:.2f} km")
print(f"Total NYC area: {total_area:.2f} km²")
```

```
Mean distance to Manhattan: 18.06 km
Maximum distance to Manhattan: 35.51 km
Total NYC area: 1363.75 km²
```

16.9. Visualizing Geospatial Data

GeoPandas integrates seamlessly with Matplotlib to create publication-quality maps and visualizations. Understanding how to effectively visualize spatial data is crucial for exploring patterns and communicating results.

16.9.1. Setting Up Plotting Environment

For crisp, professional-looking plots, configure Matplotlib with appropriate settings:

```
import matplotlib.pyplot as plt

# Set high resolution for better quality plots
plt.rcParams["figure.dpi"] = 150
```


16.9.2. Thematic Mapping

Thematic maps use visual variables like color to represent data values across geographic areas. This is one of the most effective ways to reveal spatial patterns:

```
# Create a choropleth map showing borough areas
fig, ax = plt.subplots(figsize=(10, 6))

gdf.plot(
    column="area_km2",
    ax=ax,
    legend=True,
    cmap="YlOrRd", # Yellow-Orange-Red colormap
    edgecolor="black",
    linewidth=0.5,
)

plt.title("NYC Boroughs by Area (km²)", fontsize=16, fontweight="bold")
plt.axis("off") # Remove coordinate axes for cleaner appearance
plt.tight_layout()
plt.show()
```

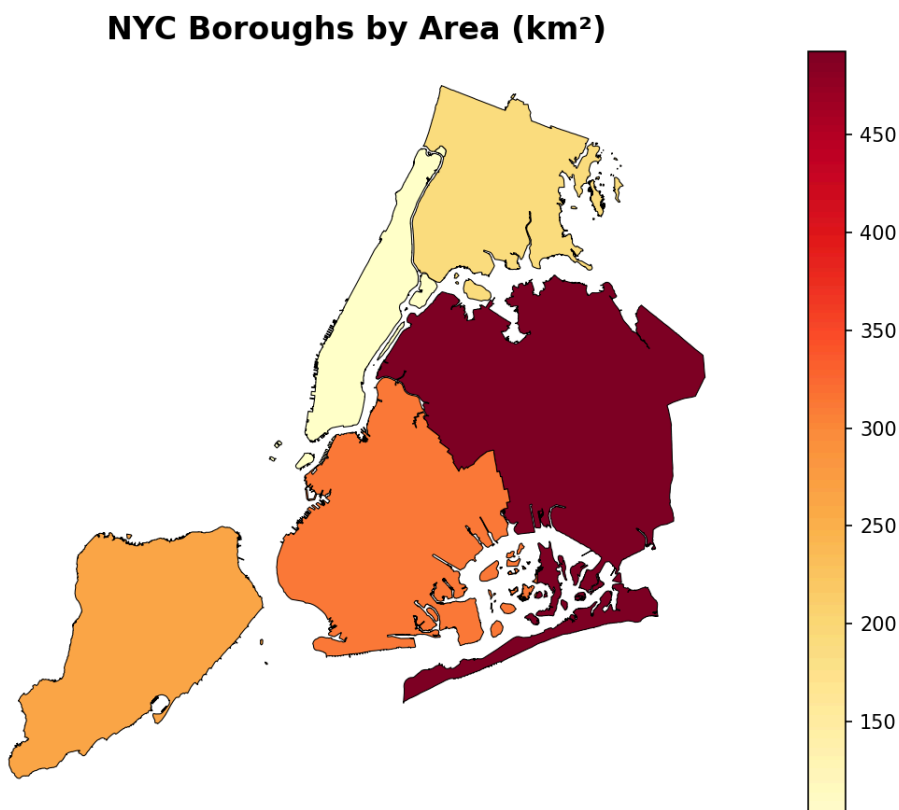


Figure 13: A choropleth map showing New York boroughs colored by their area in square kilometers.

The resulting map (Figure 13) immediately reveals spatial patterns—Queens and Brooklyn dominate in terms of area, while Manhattan appears surprisingly small despite its density and economic importance.

16.9.3. Multi-Layer Visualization

Combining different geometric elements creates more informative visualizations (Figure 14):

```
# Create a comprehensive map with multiple layers
fig, ax = plt.subplots(figsize=(10, 6))

# Plot borough boundaries as base layer
gdf["geometry"].plot(
    ax=ax, color="lightblue", edgecolor="navy", linewidth=1.5, alpha=0.7
)

# Add centroids as point layer
gdf["centroid"].plot(
    ax=ax, color="red", markersize=80, edgecolor="darkred", linewidth=1
)

# Add borough labels
for idx, row in gdf.iterrows():
    # Get centroid coordinates for label placement
    x = row.centroid.x
    y = row.centroid.y
    ax.annotate(
        idx,
        (x, y),
        xytext=(5, 5),
        textcoords="offset points",
        fontsize=10,
        fontweight="bold",
        bbox=dict(boxstyle="round,pad=0.3", facecolor="white", alpha=0.8),
    )

plt.title("NYC Borough Boundaries and Centroids", fontsize=16, fontweight="bold")
plt.axis("off")
plt.tight_layout()
plt.show()
```

NYC Borough Boundaries and Centroids



Figure 14: A multi-layer map showing NYC borough boundaries, centroids, and labels.

16.9.4. Interactive Visualization

For exploratory analysis, interactive maps provide superior user experience compared to static plots:

```
# Create an interactive map using Folium integration
m = gdf.explore(
    column="area_km2",
    cmap="YlOrRd",
    tooltip=["area_km2", "distance_to_manhattan_km"],
    popup=True,
    legend=True,
)
m
```

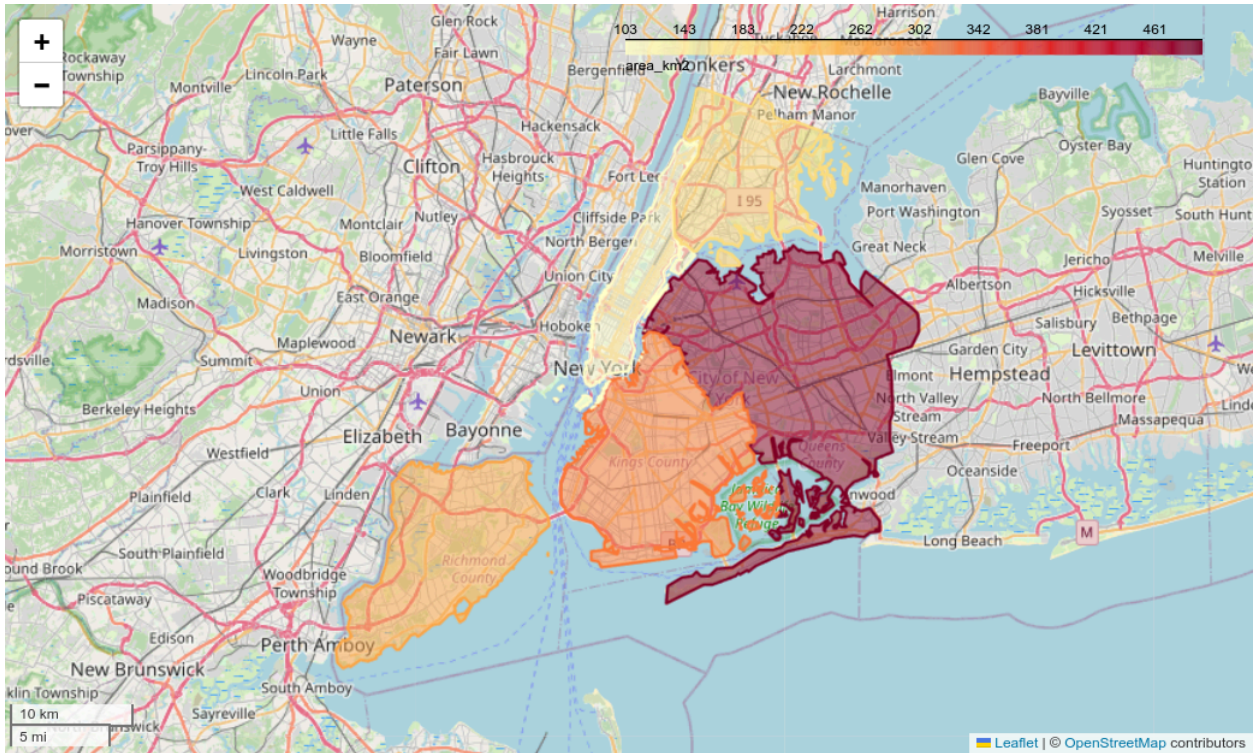


Figure 15: An interactive map of NYC borough boundaries with hover tooltips showing area and distance information.

The `explore()` method creates interactive maps with tooltips, zoom capabilities, and base map options, making it ideal for data exploration and presentation. Zooming in and out of the map (Figure 15) reveals the borough boundaries, and hovering over a borough shows its area and distance to Manhattan. Clicking on a borough opens a popup with detailed information.

16.10. Advanced Geometric Operations

GeoPandas provides sophisticated tools for geometric manipulation and analysis, enabling complex spatial analysis workflows.

16.10.1. Buffer Analysis

Buffering creates zones around geometric features, useful for proximity analysis, service area mapping, and impact assessment:

```
# Create 3-kilometer buffer zones around each borough
buffer_distance = 3000 # meters
gdf["buffered"] = gdf.buffer(buffer_distance)

print(f"Created {buffer_distance/1000} km buffer zones around each borough")
```

Visualizing original shapes alongside their buffers reveals the expanded area of influence (Figure 16):

```

# Visualize original vs buffered geometries
fig, ax = plt.subplots(figsize=(10, 6))

# Plot buffered areas first (background)
gdf["buffered"].plot(
    ax=ax,
    alpha=0.3,
    color="orange",
    edgecolor="red",
    linewidth=1,
    label="3km Buffer Zone",
)

# Plot original geometries on top
gdf["geometry"].plot(
    ax=ax,
    color="lightblue",
    edgecolor="navy",
    linewidth=1.5,
    label="Original Boundaries",
)

plt.title("NYC Boroughs: Original vs 3km Buffer Zones", fontsize=16,
fontweight="bold")
plt.legend(loc="upper right")
plt.axis("off")
plt.tight_layout()
plt.show()

```

Buffer analysis is particularly valuable for:

- **Emergency planning:** Defining evacuation zones around hazardous facilities
- **Service accessibility:** Determining areas within walking distance of facilities
- **Environmental analysis:** Studying impact zones around pollution sources

16.10.2. Convex Hull Analysis

Convex hulls represent the smallest convex shape that can enclose a geometry, useful for shape analysis and spatial indexing:

```

# Calculate convex hulls for each borough
gdf["convex_hull"] = gdf.convex_hull

# Compare areas between original shapes and convex hulls
gdf["convex_hull_area"] = gdf["convex_hull"].area / 1_000_000 # Convert to km²
gdf["area_ratio"] = gdf["convex_hull_area"] / gdf["area_km2"]

```

```
print("Convex Hull Analysis:")
print(gdf[["area_km2", "convex_hull_area", "area_ratio"]].round(2))
```

NYC Boroughs: Original vs 3km Buffer Zones

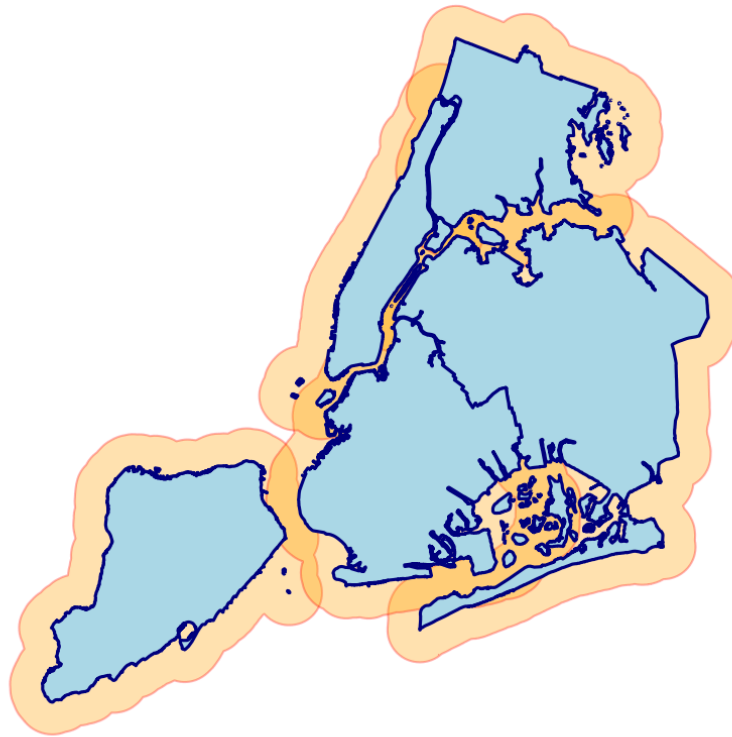


Figure 16: A comparison map showing original NYC borough boundaries and their 3-kilometer buffer zones.

Visualize the relationship between original shapes and their convex hulls (Figure 17):

```
# Create comparison visualization
fig, ax = plt.subplots(figsize=(10, 6))

# Plot original geometries
gdf["geometry"].plot(
    ax=ax, color="lightblue", edgecolor="navy", linewidth=2, label="Original Shape"
)

# Plot convex hulls as outlines
gdf["convex_hull"].plot(
    ax=ax,
```

```

    facecolor="none",
    edgecolor="red",
    linewidth=2,
    linestyle="--",
    label="Convex Hull",
)

plt.title(
    "NYC Boroughs: Original Shapes vs Convex Hulls", fontsize=16,
    fontweight="bold"
)
plt.legend(loc="upper right")
plt.axis("off")
plt.tight_layout()
plt.show()

```

NYC Boroughs: Original Shapes vs Convex Hulls

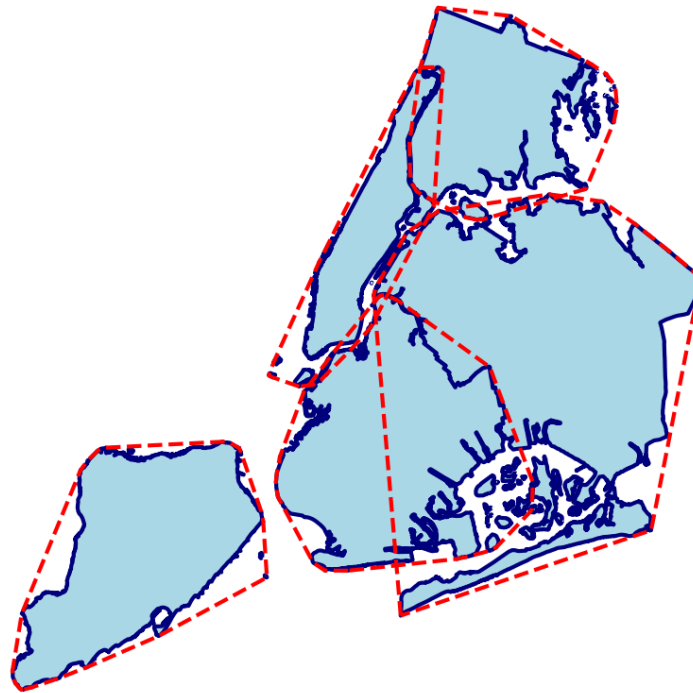


Figure 17: A comparison showing original NYC borough boundaries overlaid with their convex hulls (dashed red lines).

The area ratio reveals shape complexity—values close to 1.0 indicate relatively simple, convex shapes, while smaller ratios suggest more complex, irregular boundaries.

16.11. Spatial Relationships and Queries

Understanding spatial relationships between different geographic features is fundamental to geospatial analysis. GeoPandas provides intuitive methods for examining how geometries relate to each other.

16.11.1. Intersection Analysis

Spatial intersections help answer questions like “Which areas overlap?” or “What falls within a specific zone?”:

```
# Test which buffered boroughs intersect with Manhattan's original boundary
manhattan_geom = gdf.loc["Manhattan", "geometry"]

gdf["intersects_manhattan"] = gdf["buffered"].intersects(manhattan_geom)
gdf["touches_manhattan"] = gdf["geometry"].touches(manhattan_geom)

# Display results
intersection_results = gdf[["intersects_manhattan", "touches_manhattan"]]
intersection_results
```

This analysis reveals both direct adjacency (touches) and proximity relationships (buffer intersections) between boroughs.

16.11.2. Containment and Spatial Validation

Spatial containment tests verify expected relationships and validate data quality:

```
# Verify that centroids fall within their respective borough boundaries
gdf["centroid_within_borough"] = gdf["centroid"].within(gdf["geometry"])

# Check for any anomalies
anomalies = gdf[~gdf["centroid_within_borough"]]
if len(anomalies) > 0:
    print("Warning: Some centroids fall outside their borough boundaries")
    print(anomalies.index.tolist())
else:
    print("✓ All centroids correctly fall within their borough boundaries")
```

```
✓ All centroids correctly fall within their borough boundaries
```

This type of validation is crucial for data quality assurance in geospatial analysis workflows.

16.12. Best Practices and Performance Considerations

As your geospatial analyses become more complex, following best practices ensures efficient, reliable results.

16.12.1. Coordinate System Management

- **Always check CRS** when loading new datasets
- **Use appropriate projections** for your analysis type (geographic for global visualization, projected for measurements)
- **Document coordinate systems** in your analysis workflow

16.12.2. Memory and Performance

- **Filter data early** to reduce memory usage with large datasets
- **Use spatial indexing** for operations involving many geometric relationships
- **Choose appropriate geometric precision** for your analysis needs

16.12.3. Data Validation

- **Verify geometric validity** using `gdf.is_valid`
- **Check for empty geometries** before performing operations
- **Validate spatial relationships** as demonstrated above

16.13. Key Takeaways

This chapter introduced the fundamental concepts and operations of GeoPandas for vector data analysis. The key principles to remember:

Unified Data Model: GeoPandas extends pandas to handle geometric data alongside attributes, providing a familiar interface for spatial analysis.

Coordinate System Awareness: Understanding and properly managing coordinate reference systems is crucial for accurate spatial analysis and measurements.

Rich Spatial Operations: From basic measurements (area, distance) to complex geometric operations (buffers, intersections), GeoPandas provides comprehensive spatial analysis capabilities.

Integrated Visualization: The seamless integration with Matplotlib and interactive mapping tools makes GeoPandas ideal for both exploratory analysis and publication-quality visualization.

Scalable Workflow: The pandas-like interface ensures that GeoPandas skills scale from simple analyses to complex, multi-dataset spatial workflows.

GeoPandas bridges the gap between traditional GIS software and modern data science workflows, enabling geospatial analysis within the broader Python ecosystem. This integration makes it an invaluable tool for anyone working with location-based data, from urban planners and environmental scientists to data analysts and researchers.

16.14. Exercises

16.14.1. Exercise 1: Creating and Manipulating GeoDataFrames with GeoPandas

This exercise focuses on creating and manipulating GeoDataFrames, performing spatial operations, and visualizing the data.

1. Load the New York City building dataset from the GeoJSON file using GeoPandas: https://github.com/opengeos/datasets/releases/download/places/nyc_buildings.geojson
2. Create a plot of the building footprints and color them based on the building height (use the `height_MS` column).
3. Create an interactive map of the building footprints and color them based on the building height (use the `height_MS` column).
4. Calculate the average building height (use the `height_MS` column).
5. Select buildings with a height greater than the average height.
6. Save the GeoDataFrame to a new GeoJSON file.

16.14.2. Exercise 2: Combining NumPy, Pandas, and GeoPandas

This exercise requires you to combine the power of NumPy, Pandas, and GeoPandas to analyze and visualize spatial data.

1. Use Pandas to load the world cities dataset from this URL: https://github.com/opengeos/datasets/releases/download/world/world_cities.csv
2. Filter the dataset to include only cities with latitude values between -40 and 60 (i.e., cities located in the Northern Hemisphere or near the equator).
3. Create a GeoDataFrame from the filtered dataset by converting the latitude and longitude into geometries.
4. Reproject the GeoDataFrame to the Mercator projection (EPSG:3857).
5. Calculate the distance (in meters) between each city and the city of Paris.
6. Plot the cities on a world map, coloring the points by their distance from Paris.

Chapter 17. Working with Raster Data Using Rasterio

17.1. Introduction

Rasterio⁵⁰ is a Python library that allows you to read, write, and analyze geospatial raster data. Built on top of **GDAL**⁵¹ (Geospatial Data Abstraction Library), it provides an efficient interface to work with raster datasets, such as satellite images, digital elevation models (DEMs), and other gridded data. Rasterio simplifies common geospatial tasks and helps to bridge the gap between raw geospatial data and analysis, especially when combined with other Python libraries like `numpy`, `pandas`, and `matplotlib`.

17.1.1. What is Raster Data?

Raster data is a fundamental type of geospatial data that represents the world as a grid of pixels (cells). Each pixel in the grid contains a value that represents some geographic information, such as:

- Elevation (in digital elevation models)
- Temperature (in climate data)
- Reflectance values (in satellite imagery)
- Land cover types (in classification maps)

The key characteristics of raster data are:

1. **Spatial Resolution:** The size of each pixel in real-world units (e.g., 30 meters)
2. **Extent:** The geographic area covered by the raster
3. **Coordinate Reference System (CRS):** How the pixel grid relates to real-world coordinates
4. **Number of Bands:** Some rasters have multiple bands (like RGB images), while others have a single band (like elevation data)

17.1.2. Why Use Rasterio?

Rasterio provides several advantages for working with raster data:

- **Memory Efficiency:** It can handle large datasets by reading data in chunks
- **Geospatial Awareness:** It maintains the spatial reference information of your data
- **Integration:** It works seamlessly with other Python scientific libraries
- **Performance:** It's built on GDAL, which is highly optimized for geospatial operations

17.2. Learning Objectives

By the end of this chapter, you should be able to:

- Understand the basic concepts of raster data and how Rasterio handles them
- Read and write raster datasets while preserving their geospatial properties
- Extract and interpret raster metadata and perform basic operations on raster bands
- Visualize raster data using appropriate color schemes and overlays
- Perform fundamental geospatial operations like clipping and basic band math

⁵⁰<https://rasterio.readthedocs.io>

⁵¹<https://gdal.org>

17.3. Installing Rasterio

Before working with Rasterio, you need to install the library. You can do this by running the following command in your Python environment. Uncomment the line below if you're working in a Jupyter notebook or other interactive Python environment.

```
# %pip install rasterio pygis
```

To get started, you'll need to import rasterio along with a few other useful Python libraries. These libraries will allow us to perform different types of geospatial operations, manipulate arrays, and visualize raster data.

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

- `rasterio` : The main library for reading and writing raster data.
- `rasterio.plot` : A submodule of Rasterio for plotting raster data.
- `geopandas` : A popular library for handling vector geospatial data.
- `numpy` : A powerful library for array manipulations, which is very useful for raster data.
- `matplotlib` : A standard plotting library in Python for creating visualizations.

17.4. Reading Raster Data

17.4.1. Opening Raster Files

To read raster data, you can use the `rasterio.open()` function. This function creates a connection to the file without loading the entire dataset into memory. This is particularly useful for large datasets like satellite imagery or high-resolution DEMs, as they might be too big to fit into memory at once.

The `rasterio.open()` function returns a `DatasetReader` object that provides access to the raster's data and metadata. This object is your gateway to working with the raster data, allowing you to:

- Read pixel values
- Access metadata
- Perform operations on the data
- Visualize the raster

Here's a simple example of opening a DEM (digital elevation model) raster file:

```
raster_path = (
    "https://github.com/opengeos/datasets/releases/download/raster/dem_90m.tif"
)
src = rasterio.open(raster_path)
print(src)
```

17.4.2. Understanding Raster Metadata

When working with raster data, it's crucial to understand its metadata - the information that describes the raster's properties. Rasterio provides several ways to access this information:

17.4.2.1. Basic File Information

1. File Name and Mode:

- The `name` attribute gives you the file path or URL
- The `mode` attribute shows how the file was opened (read-only 'r' or write 'w')

```
print(f"File name: {src.name}")
print(f"File mode: {src.mode}")
```

2. Complete Metadata: The `meta` attribute provides a dictionary containing all the essential raster properties:

```
print("Raster metadata:")
for key, value in src.meta.items():
    print(f"{key}: {value}")
```

```
Raster metadata:
driver: GTiff
dtype: int16
nodata: None
width: 4269
height: 3113
count: 1
crs: EPSG:3857
transform: | 90.00, 0.00, -13442488.34|
| 0.00, -90.00, 4668371.58|
| 0.00, 0.00, 1.00|
```

17.4.2.2. Spatial Properties

1. Coordinate Reference System (CRS): The CRS defines how the raster's pixel coordinates relate to real-world locations. It's essential for:

- Understanding the spatial context of your data
- Ensuring proper alignment with other geospatial data
- Performing accurate spatial analysis

```
print(f"Coordinate Reference System: {src.crs}")
```

2. Spatial Resolution: Resolution tells you the size of each pixel in real-world units (e.g., meters). It's crucial for:

- Understanding the level of detail in your data
- Determining the appropriate scale for analysis

- Comparing different raster datasets

```
print(f"Pixel size (x, y): {src.res}")
```

3. **Raster Dimensions:** The width and height tell you the number of pixels in each dimension:

```
print(f"Raster dimensions: {src.width} x {src.height} pixels")
```

4. **Geographic Extent:** The `bounds` attribute provides the coordinates of the raster's edges:

```
print(f"Geographic bounds: {src.bounds}")
```

17.4.2.3. Data Properties

1. **Data Types:** The `dtypes` attribute tells you the data type of the pixel values. This is important for:
- Understanding the range of possible values
 - Performing mathematical operations
 - Managing memory usage

```
print(f"Data types: {src.dtypes}")
```

2. **Number of Bands:** Some rasters have multiple bands (like RGB images), while others have a single band (like elevation data):

```
print(f"Number of bands: {src.count}")
```

17.4.3. The Affine Transform

The affine transform is a mathematical relationship that maps pixel coordinates to geographic coordinates. It's represented by six parameters that control:

- Pixel size (scale)
- Rotation
- Translation (position)

```
print("Affine transform:")  
print(src.transform)
```

```
Affine transform:  
| 90.00, 0.00, -13442488.34|  
| 0.00, -90.00, 4668371.58|  
| 0.00, 0.00, 1.00|
```

The transform parameters are:

- `a` : width of a pixel in the x-direction
- `b` : row rotation (typically zero)
- `c` : x-coordinate of the upper-left corner
- `d` : column rotation (typically zero)
- `e` : height of a pixel in the y-direction (typically negative)
- `f` : y-coordinate of the upper-left corner

Understanding these parameters is essential for:

- Converting between pixel and geographic coordinates
- Performing accurate spatial operations
- Maintaining the spatial integrity of your data

17.5. Visualizing Raster Data

Visualization is a crucial part of working with raster data. It helps you:

- Understand the spatial patterns in your data
- Identify potential issues or anomalies
- Communicate your findings effectively

Rasterio integrates with Matplotlib to provide powerful visualization capabilities. Let's explore the different ways to visualize raster data.

17.5.1. Basic Raster Visualization

The simplest way to visualize a raster is using `rasterio.plot.show()`. This function automatically handles the raster's georeferencing, ensuring that the data is displayed in its correct geographic position (see [Figure 18](#)).

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

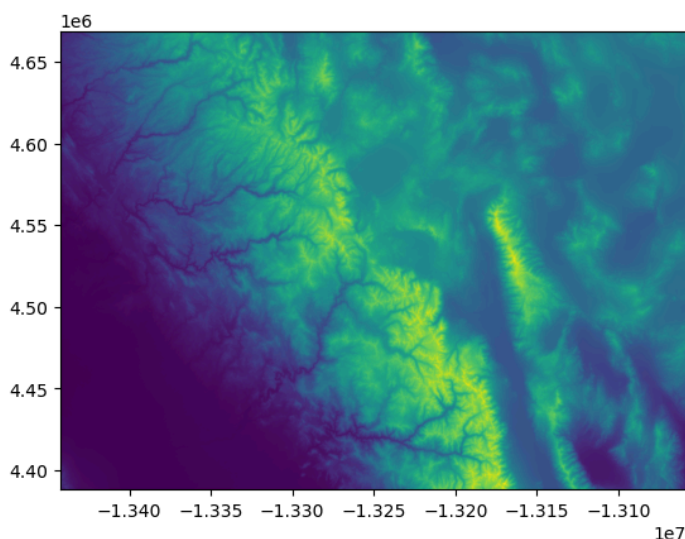


Figure 18: Visualization of a Digital Elevation Model (DEM) using Rasterio.

17.5.2. Understanding Color Maps

Color maps (colormaps) are essential for effectively visualizing raster data. They map the pixel values to colors, making it easier to interpret the data. Common colormaps include:

- `'terrain'`: Good for elevation data, showing low to high values with intuitive colors
- `'viridis'`: A perceptually uniform colormap, good for general use
- `'gray'` or `'Greys'`: Useful for visualizing single-band data
- `'RdYlBu'`: Red-Yellow-Blue, good for showing positive and negative values

Here's an example using the 'terrain' colormap for our DEM (see [Figure 19](#)):

```
fig, ax = plt.subplots(figsize=(8, 8))
rasterio.plot.show(src, cmap="terrain", ax=ax, title="Digital Elevation Model (DEM)")
plt.show()
```

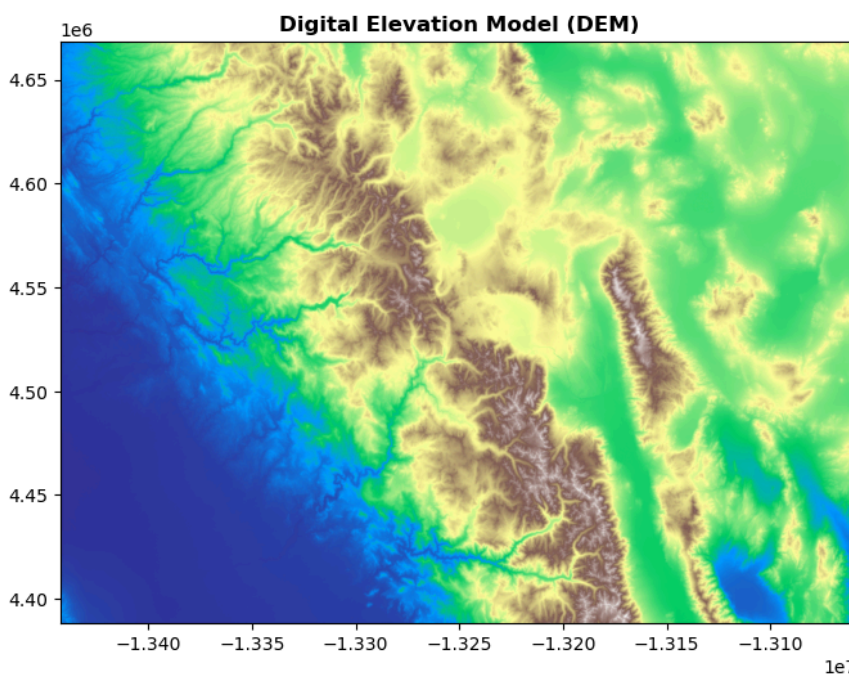


Figure 19: Visualization of a Digital Elevation Model (DEM) using Rasterio with a colormap.

17.5.3. Adding Colorbars

Colorbars are essential for interpreting the values in your visualization. They show the relationship between colors and values in your data. The `rasterio.plot.show()` function does not automatically add a colorbar, but you can add one manually using the `matplotlib` library (see [Figure 20](#)):

```
elev_band = src.read(1)
plt.figure(figsize=(8, 8))
plt.imshow(elev_band, cmap="terrain")
```



```
plt.colorbar(label="Elevation (meters)", shrink=0.5)
plt.title("DEM with Terrain Colormap")
plt.show()
```

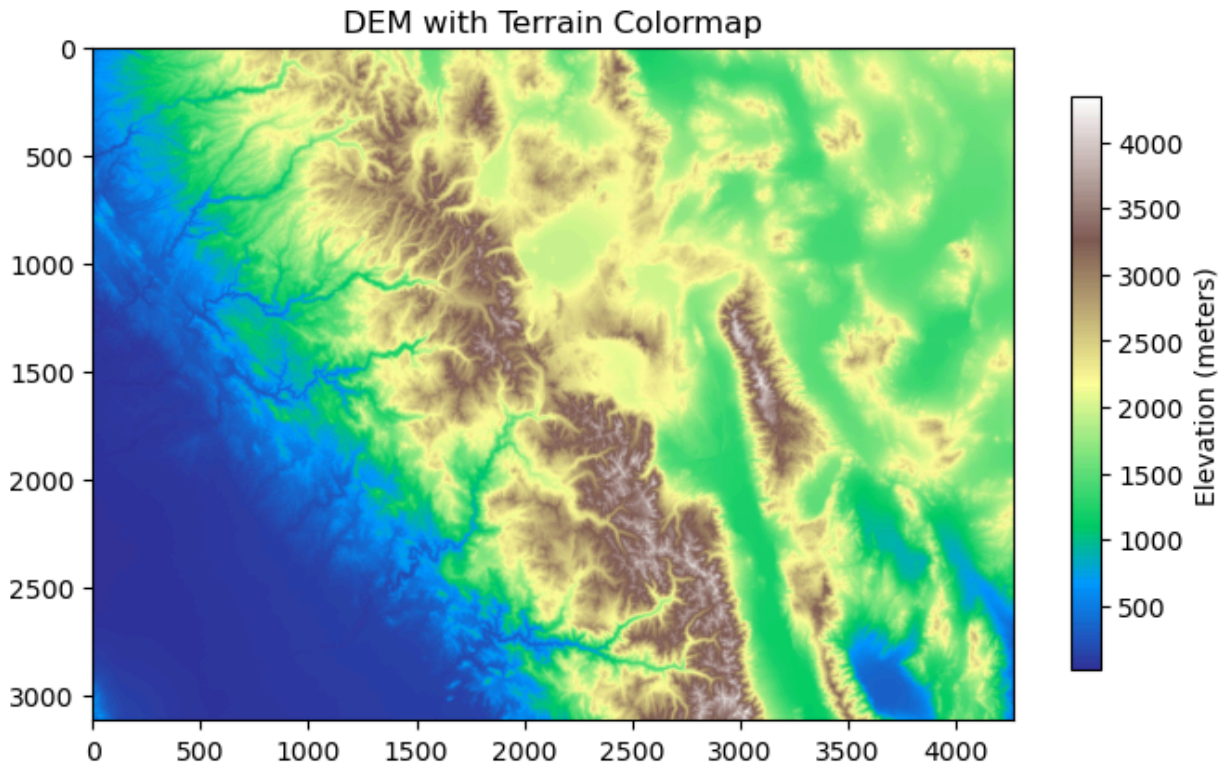


Figure 20: Visualization of a raster dataset with a colorbar.

17.5.4. Visualizing Multiple Bands

When working with multi-band rasters (like satellite imagery), you can visualize individual bands or combine them to create composite images. Let's use a Landsat 9 image for demonstration.

```
raster_path = "https://github.com/opengeos/datasets/releases/download/raster/LC
09_039035_20240708_90m.tif"
src = rasterio.open(raster_path)
```

17.5.4.1. Single Band Visualization

To visualize a specific band, specify the band number (1-based indexing). For example, to visualize the first band of the raster dataset (see [Figure 21](#)):

```
rasterio.plot.show((src, 1), cmap="gray", title="Band 1")
```

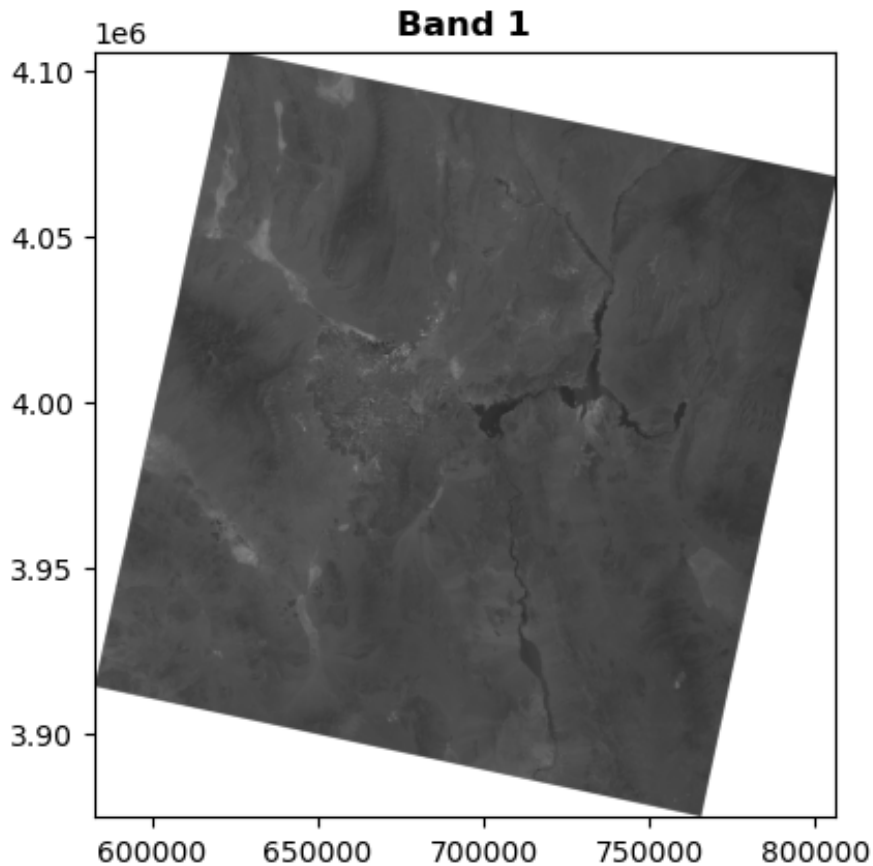


Figure 21: Visualization of a single band of a raster dataset using Rasterio.

17.5.4.2. RGB Composite

For satellite imagery, you can create a true-color composite by combining the red, green, and blue bands (see [Figure 22](#)):

```
nir_band = src.read(5)
red_band = src.read(4)
green_band = src.read(3)

# Stack the bands into a single array
rgb = np.dstack((nir_band, red_band, green_band)).clip(0, 1)

# Plot the stacked array
plt.figure(figsize=(8, 8))
plt.imshow(rgb)
plt.title("Bands NIR, Red, and Green combined")
plt.show()
```

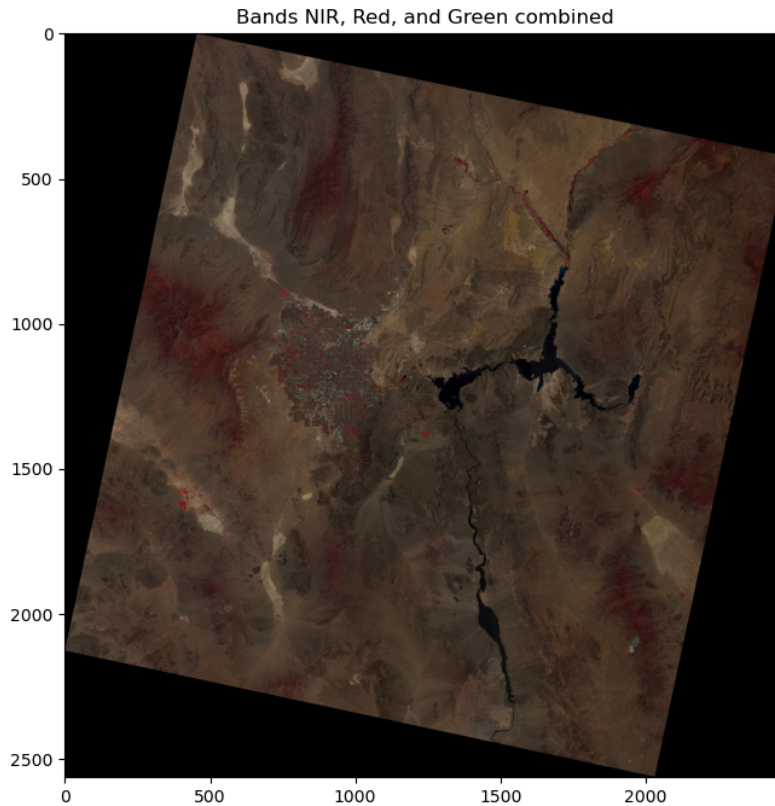


Figure 22: Visualization of RGB bands of a raster dataset using Rasterio.

17.5.4.3. Creating a Multi-Panel Plot

To visualize all the bands together, we can create a multi-panel plot, displaying each band with its respective name (see [Figure 23](#)):

```
band_names = ["Coastal Aerosol", "Blue", "Green", "Red", "NIR", "SWIR1", "SWIR2"]

fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(8, 10))
axes = axes.flatten() # Flatten the 2D array of axes to 1D for easy iteration

for band in range(1, src.count):
    data = src.read(band)
    ax = axes[band - 1]
    im = ax.imshow(data, cmap="gray", vmin=0, vmax=0.5)
    ax.set_title(f"Band {band_names[band - 1]}")
    fig.colorbar(im, ax=ax, label="Reflectance", shrink=0.5)

plt.tight_layout()
plt.show()
```

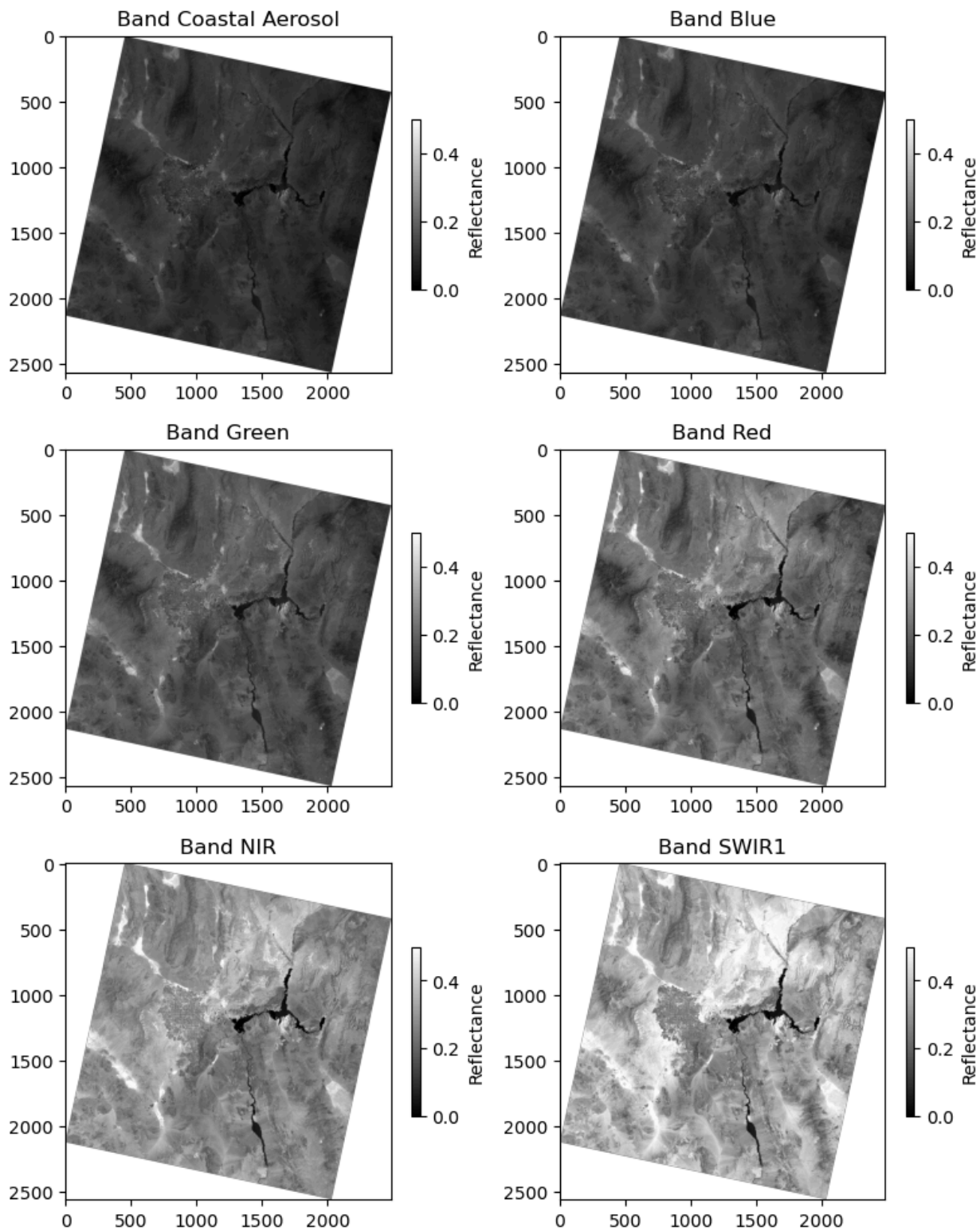


Figure 23: Visualization of a raster dataset using Rasterio with multiple panels.

17.5.5. Overlaying Vector Data

Often, you'll want to overlay vector data (like boundaries or points) on top of your raster visualization. This helps provide context and additional information. Below is an example of overlaying a vector dataset on top of a raster dataset (see [Figure 24](#)):

```
# Load raster data
raster_path = (
    "https://github.com/opegeos/datasets/releases/download/raster/dem_90m.tif"
)
src = rasterio.open(raster_path)

# Load vector data
vector_path = (
    "https://github.com/opegeos/datasets/releases/download/places/dem_bounds.
    geojson"
)
gdf = gpd.read_file(vector_path)
gdf = gdf.to_crs(src.crs) # Ensure same CRS as raster

# Create the plot
fig, ax = plt.subplots(figsize=(8, 8))
rasterio.plot.show(src, cmap="terrain", ax=ax, title="DEM with Vector Overlay")
gdf.plot(ax=ax, edgecolor="red", facecolor="none", linewidth=2)
plt.show()
```

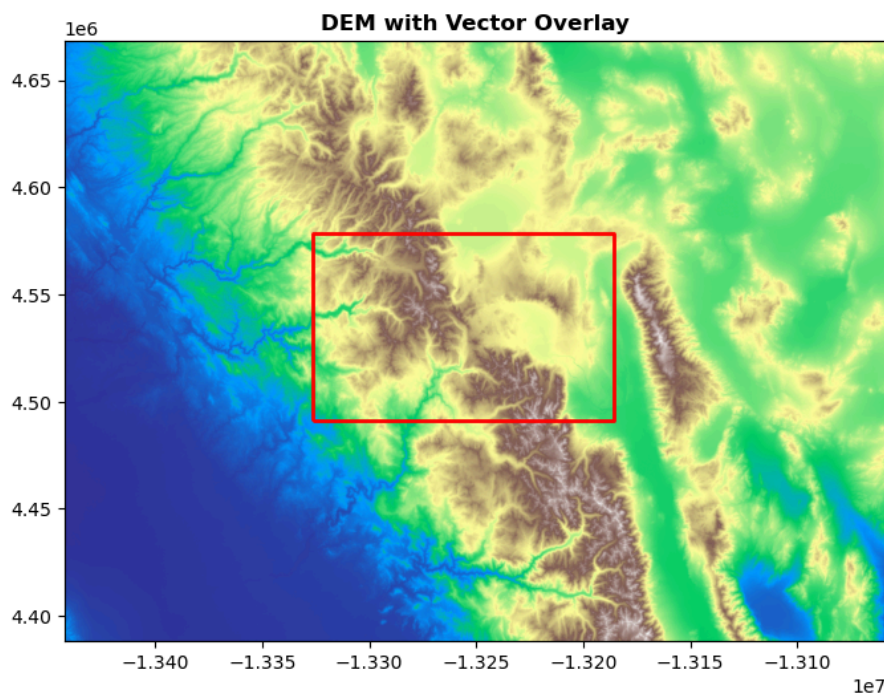


Figure 24: Overlaying vector data on a raster dataset using Rasterio.

17.6. Accessing and Manipulating Raster Bands

Raster datasets often consist of multiple bands, each capturing a different part of the electromagnetic spectrum. For instance, satellite images may include separate bands for red, green, blue, and near-infrared (NIR) wavelengths.

17.6.1. Stacking Multiple Bands

We can combine several bands into a single image by stacking them into an array. For example, we'll stack the NIR, Red, and Green bands:

```
raster_path = "https://github.com/opengeos/datasets/releases/download/raster/LC
09_039035_20240708_90m.tif"
src = rasterio.open(raster_path)
```

```
nir_band = src.read(5)
red_band = src.read(4)
green_band = src.read(3)

# Stack the bands into a single array
rgb = np.dstack((nir_band, red_band, green_band)).clip(0, 1)

print(rgb.shape)
```

17.6.2. Basic Band Math (NDVI Calculation)

Band math enables us to perform computations across different bands. A common application is calculating the Normalized Difference Vegetation Index (NDVI), which is an indicator of vegetation health.

NDVI is calculated as:

$$\text{NDVI} = (\text{NIR} - \text{Red}) / (\text{NIR} + \text{Red})$$

We can compute and plot the NDVI and display it as a color map (see [Figure 25](#)):

```
# NDVI Calculation: NDVI = (NIR - Red) / (NIR + Red)
ndvi = (nir_band - red_band) / (nir_band + red_band)
ndvi = ndvi.clip(-1, 1)

plt.figure(figsize=(8, 8))
plt.imshow(ndvi, cmap="RdYlGn", vmin=-1, vmax=1)
plt.colorbar(label="NDVI", shrink=0.75)
plt.title("NDVI")
plt.xlabel("Column #")
plt.ylabel("Row #")
plt.show()
```

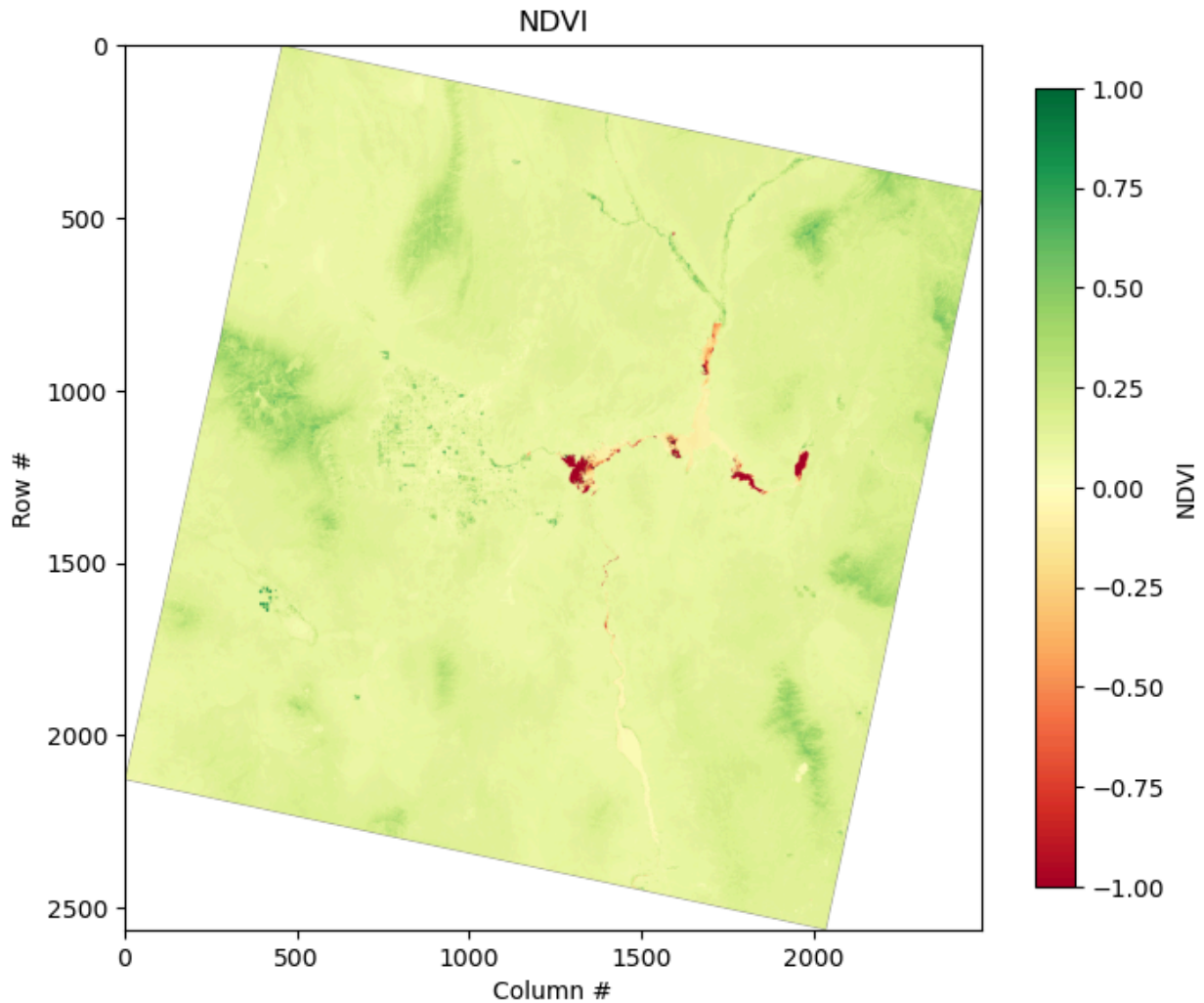



Figure 25: Visualization of NDVI (Normalized Difference Vegetation Index) using Rasterio.

17.7. Writing Raster Data

After processing the raster data (e.g., computing NDVI), you may want to save the results to a new file. Using `rasterio`, we can write the data back to a GeoTIFF file.

First, we review and update the profile (metadata) for the output file:

```
with rasterio.open(raster_path) as src:
    profile = src.profile
    print(profile)
```

Then, we adjust the profile to fit the modified dataset (e.g., NDVI) since the NDVI dataset only has one band:

```
profile.update(dtype=rasterio.float32, count=1, compress="lzw")
print(profile)
```

Finally, we write the NDVI data to a new file:

```
output_raster_path = "ndvi.tif"

with rasterio.open(output_raster_path, "w", **profile) as dst:
    dst.write(ndvi, 1)
print(f"Raster data has been written to {output_raster_path}")
```

17.8. Clipping Raster Data

17.8.1. Clipping with a Bounding Box

To extract a subset of the raster data, we can either slice the array or use geographic bounds.

First, let's open the sample raster dataset:

```
raster_path = "https://github.com/opengeos/datasets/releases/download/raster/LC
09_039035_20240708_90m.tif"
src = rasterio.open(raster_path)
data = src.read()
```

```
data.shape
```

Then, let's clip a portion of the raster data using array indices (see [Figure 26](#)):

```
subset = data[:, 900:1400, 700:1200].clip(0, 1)
rgb_subset = np.dstack((subset[4], subset[3], subset[2]))
rgb_subset.shape
```

```
# Plot the stacked array
plt.figure(figsize=(8, 8))
plt.imshow(rgb_subset)
plt.title("Las Vegas, NV")
plt.show()
```

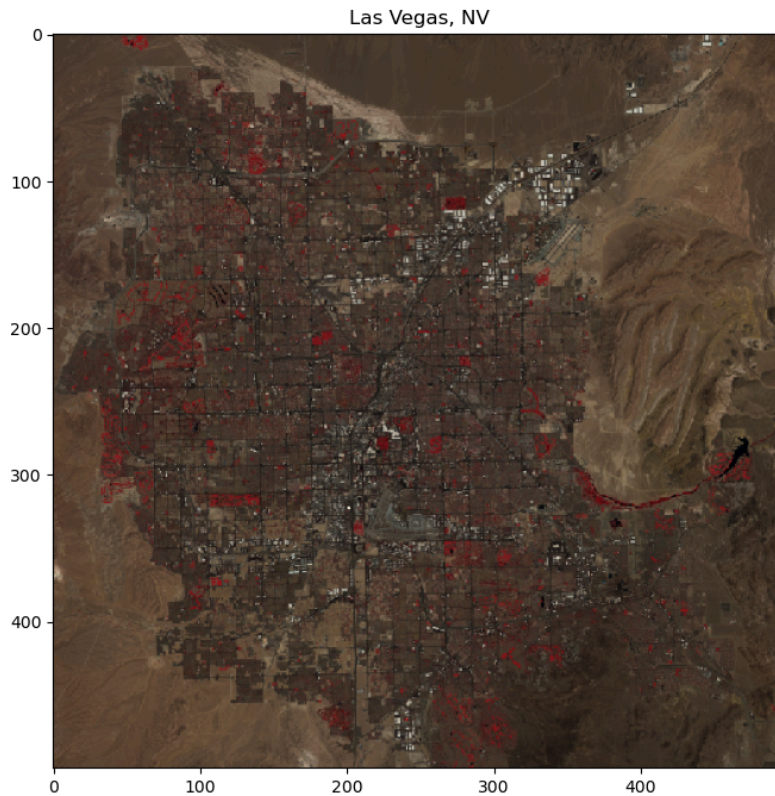



Figure 26: Clipping raster data using Rasterio.

Alternatively, we can use a specific geographic window to clip the data:

```
from rasterio.windows import Window
from rasterio.transform import from_bounds

# Assuming subset and src are already defined
# Define the window of the subset (replace with actual window coordinates)
window = Window(col_off=700, row_off=900, width=500, height=500)

# Calculate the bounds of the window
window_bounds = rasterio.windows.bounds(window, src.transform)

# Calculate the new transform based on the window bounds
new_transform = from_bounds(*window_bounds, window.width, window.height)
```

After defining the window, we write the clipped data to a new file:

```
with rasterio.open(
    "las_vegas.tif",
    "w",
    driver="GTiff",
```

```

height=subset.shape[1],
width=subset.shape[2],
count=subset.shape[0],
dtype=subset.dtype,
crs=src.crs,
transform=new_transform,
compress="lzw",
) as dst:
    dst.write(subset)

```

17.8.2. Clipping with a Vector Dataset

To clip the raster using vector data (e.g., a GeoJSON bounding box), we can use `rasterio.mask`. First, load the vector data:

```

import fiona
import rasterio.mask

```

```

geojson_path = "https://github.com/opengeos/datasets/releases/download/places/
las_vegas_bounds_utm.geojson"
bounds = gpd.read_file(geojson_path)

```

Visualize the raster and vector data together:

```

fig, ax = plt.subplots()
rasterio.plot.show(src, ax=ax)
bounds.plot(ax=ax, edgecolor="red", facecolor="none")

```

Next, apply the mask to extract only the area within the vector bounds:

```

with fiona.open(geojson_path, "r") as f:
    shapes = [feature["geometry"] for feature in f]
    out_image, out_transform = rasterio.mask.mask(src, shapes, crop=True)

```

Finally, write the clipped raster to a new file:

```

out_meta = src.meta
out_meta.update(
    {
        "driver": "GTiff",
        "height": out_image.shape[1],
        "width": out_image.shape[2],
        "transform": out_transform,
    }
)

```

```
)  
  
with rasterio.open("las_vegas_clip.tif", "w", **out_meta) as dst:  
    dst.write(out_image)
```

17.9. Key Takeaways

In this chapter, we've covered the fundamental concepts and operations for working with raster data using Rasterio. Here are the key points to remember:

1. Understanding Raster Data

- Raster data represents the world as a grid of pixels
- Each pixel contains a value representing geographic information
- Rasters can have single or multiple bands
- Spatial reference information is crucial for proper interpretation

2. Working with Rasterio

- Rasterio provides an efficient interface for reading and writing raster data
- It maintains geospatial properties of the data
- It integrates well with other Python scientific libraries
- It handles large datasets through windowed reading

3. Essential Operations

- Reading and writing raster data
- Accessing and manipulating bands
- Performing basic mathematical operations
- Visualizing raster data with appropriate colormaps
- Clipping and subsetting raster data

4. Best Practices

- Always check metadata before operations
- Use appropriate data types and compression
- Handle no-data values properly
- Maintain spatial reference information
- Close files when done

17.10. Exercises

The following exercises will help you practice the fundamental concepts covered in this chapter. Each exercise builds on the previous ones and focuses on essential raster data operations.

17.10.1. Sample Datasets

For these exercises, we'll use two sample datasets:

1. Single-band DEM (Digital Elevation Model)

- URL: https://github.com/opengeos/datasets/releases/download/raster/dem_90m.tif
- Type: Elevation data
- Resolution: 90 meters

- Bands: 1

2. Multispectral Satellite Image

- URL: <https://github.com/openeos/datasets/releases/download/raster/cog.tif>
- Type: Landsat imagery
- Resolution: 30 meters
- Bands: Multiple (including RGB and infrared)

17.10.2. Exercise 1: Reading and Exploring Raster Data

This exercise focuses on the fundamental skills of reading and understanding raster data.

1. Open the DEM raster file using `rasterio.open()`
2. Print the following information:
 - File name and mode
 - Coordinate Reference System (CRS)
 - Spatial resolution
 - Raster dimensions (width and height)
 - Data type
 - Number of bands
3. Create a simple visualization of the DEM using an appropriate colormap

17.10.3. Exercise 2: Working with Raster Bands

This exercise helps you understand how to work with different bands in a raster dataset.

1. Open the multispectral image
2. Print the number of bands and their data types
3. Create a visualization showing:
 - The first band in grayscale
 - A true-color composite (RGB) using bands 3, 2, and 1
 - Add appropriate titles and colorbars

17.10.4. Exercise 3: Basic Raster Operations

This exercise practices fundamental raster operations.

1. Using the DEM:
 - Calculate and print basic statistics (mean, min, max, standard deviation)
 - Create a histogram of the elevation values
 - Identify any potential outliers or unusual values
2. Using the multispectral image:
 - Calculate the mean value for each band
 - Create a simple band ratio (e.g., band 4 / band 3)
 - Visualize the results with appropriate colormaps

17.10.5. Exercise 4: Writing and Saving Raster Data

This exercise focuses on saving processed raster data.

1. Using the DEM:

- Calculate the slope (you can use a simple gradient)
- Save the slope as a new GeoTIFF file
- Include appropriate metadata

2. Using the multispectral image:

- Create a simple vegetation index (e.g., $(\text{band 4} - \text{band 3}) / (\text{band 4} + \text{band 3})$)
- Save the result as a new GeoTIFF file
- Include appropriate metadata

17.10.6. Exercise 5: Clipping and Subsetting

This exercise practices extracting subsets of raster data.

1. Using the DEM:

- Extract a 500x500 pixel subset from the center of the image
- Save the subset as a new file
- Visualize both the original and subset to verify the operation

2. Using the multispectral image:

- Extract a subset using a specific geographic window
- Save the subset as a new file
- Create a true-color composite of the subset

Chapter 18. Multi-dimensional Data Analysis with Xarray

18.1. Introduction

Imagine you’re analyzing climate data that varies across three dimensions: latitude, longitude, and time. With traditional tools, you might find yourself constantly keeping track of which array index corresponds to which dimension, carefully managing metadata, and writing complex loops to perform seemingly simple operations. This is where [Xarray](#)⁵² transforms your workflow from tedious to intuitive.

Xarray is a powerful Python library designed specifically for working with multi-dimensional labeled datasets—the kind of data that’s ubiquitous in geospatial analysis, climate science, oceanography, and remote sensing. Rather than working with anonymous arrays of numbers, Xarray lets you work with data that knows about itself: its dimensions have meaningful names (like “time”, “latitude”, “longitude”), its coordinates have real values (like dates and geographic locations), and it carries metadata that describes what the data represents.

Why Xarray revolutionizes multi-dimensional data analysis:

Consider a typical climate dataset containing temperature measurements. With raw NumPy arrays, selecting data for a specific date might require remembering that time is the first dimension and finding the right index. With Xarray, you simply write `temperature.sel(time='2023-06-15')` and the library handles the rest. This intuitive approach extends to all operations, making complex analyses readable and maintainable.

Key advantages of Xarray:

- **Self-describing data:** Dimensions, coordinates, and metadata travel with your data
- **Intuitive indexing:** Select data using meaningful labels rather than numeric indices
- **Broadcasting and alignment:** Automatic handling of operations between arrays with different dimensions
- **Integration ecosystem:** Seamless interoperability with pandas, NumPy, Matplotlib, and other scientific Python tools
- **Performance optimization:** Built-in support for lazy evaluation and parallel computing through [Dask](#)⁵³
- **Format flexibility:** Native support for scientific data formats like [NetCDF](#)⁵⁴ and [Zarr](#)⁵⁵

18.2. Learning Objectives

By the end of this chapter, you should be able to:

- Understand the fundamental concepts and data structures in Xarray: `DataArray` and `Dataset`
- Load and inspect multi-dimensional geospatial datasets using Xarray’s built-in and external data sources
- Perform intuitive data selection and indexing using both label-based and position-based methods
- Execute basic operations on Xarray objects, including arithmetic operations and statistical calculations
- Visualize multi-dimensional data using Xarray’s integrated plotting capabilities
- Apply Xarray to common geospatial analysis tasks such as time series analysis and spatial statistics
- Read and write data in scientific formats like NetCDF for reproducible research workflows

⁵²<https://docs.xarray.dev>

⁵³<https://www.dask.org>

⁵⁴<https://www.unidata.ucar.edu/software/netcdf>

⁵⁵<https://zarr.dev>

18.3. Understanding Xarray's Data Model

Before diving into practical examples, it's essential to understand Xarray's data model (Figure 27), which consists of two primary structures that work together to represent complex, multi-dimensional datasets.

18.3.1. Core Data Structures

DataArray: Think of a DataArray as a NumPy array that has learned to speak—it knows what its dimensions represent, where its data points are located in space and time, and what the data actually measures. A DataArray consists of:

- **Data values:** The actual numeric data (temperature readings, precipitation amounts, etc.)
- **Dimensions:** Named axes that give meaning to each array dimension (e.g., 'time', 'lat', 'lon')
- **Coordinates:** The actual coordinate values along each dimension (e.g., specific dates, latitude values, longitude values)
- **Attributes:** Metadata describing the data (units, description, data source, etc.)

Dataset: A Dataset is a collection of DataArrays that share the same dimensions and coordinates. Think of it as a container that holds related variables—like a climate dataset containing both temperature and precipitation data measured at the same locations and times.

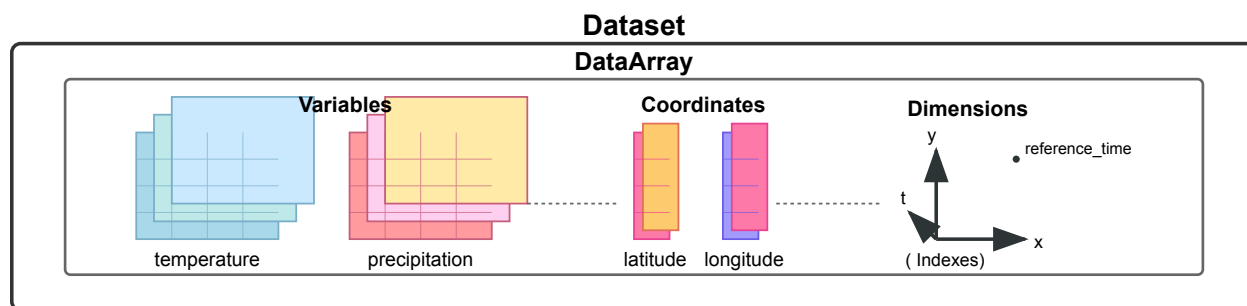


Figure 27: Xarray's data model.

This structure mirrors how we naturally think about scientific data: we have measurements (DataArrays) taken at specific locations and times (coordinates) that we want to analyze together (Dataset).

18.3.2. Why This Structure Matters

Traditional array-based approaches require you to remember that “dimension 0 is time, dimension 1 is latitude, dimension 2 is longitude.” With Xarray, you work directly with meaningful names, making your code self-documenting and much less error-prone. Operations automatically align data based on coordinates, and metadata travels with your data through all transformations.

18.4. Setting Up Your Environment

Before we start exploring Xarray's capabilities, let's ensure you have the necessary libraries installed and properly configured.

18.4.1. Installing Required Packages

Xarray integrates with several other libraries to provide its full functionality:

```
%pip install xarray pooch pygis
```

18.4.2. Importing Libraries and Configuration

```
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

# Configure Xarray for better display and performance
xr.set_options(keep_attrs=True, display_expand_data=False)

# Configure NumPy display for cleaner output
np.set_printoptions(threshold=10, edgeitems=2)

# Configure matplotlib for better plots
plt.rcParams["figure.dpi"] = 150
```

Configuration explanations:

- `keep_attrs=True` : Preserves metadata attributes through operations
- `display_expand_data=False` : Shows a compact representation of large datasets
- NumPy settings: Prevents overwhelming output when displaying large arrays

18.5. Loading and Exploring Real Climate Data

Let's begin with a practical example using real climate data. Xarray provides several built-in tutorial datasets that demonstrate common data structures and analysis patterns.

18.5.1. Loading Tutorial Data

Xarray includes access to several [tutorial datasets](https://tinyurl.com/xarray-tutorial-dataset)⁵⁶ that represent typical scientific data structures. We'll start with an air temperature dataset ([Figure 28](#)).

```
# Load a climate dataset with air temperature measurements
ds = xr.tutorial.open_dataset("air_temperature")
ds
```

This dataset represents a common structure in climate science: temperature measurements (in Kelvin) recorded at weather stations across North America over time. The data is stored in NetCDF format, a self-describing scientific data format that preserves all the metadata and coordinate information.

Understanding the output: When you display a Dataset, Xarray shows you a wealth of information:

- **Dimensions:** The named axes and their sizes
- **Coordinates:** The actual coordinate values along each dimension
- **Data variables:** The measured quantities (in this case, air temperature)

⁵⁶<https://tinyurl.com/xarray-tutorial-dataset>

- **Attributes:** Global metadata describing the entire dataset

Data source and caching: The dataset is automatically downloaded from the internet and cached locally. You can find the cache directory at:

- **Linux:** `~/.cache/xarray_tutorial_data`
- **macOS:** `~/Library/Caches/xarray_tutorial_data`
- **Windows:** `~/AppData/Local/xarray_tutorial_data`

This caching ensures you don't re-download data unnecessarily and enables offline work after the initial download.

xarray.Dataset

► Dimensions: (lat: 25, time: 2920, lon: 53)

▼ Coordinates:

lat	(lat)	float32	75.0 72.5 70.0 ... 20.0 17.5 15.0	 
lon	(lon)	float32	200.0 202.5 205.0 ... 327.5 330.0	 
time	(time)	datetime64[ns]	2013-01-01 ... 2014-12-31T18:00:00	 

▼ Data variables:

air	(time, lat, lon)	float64	...	 
------------	------------------	---------	-----	---

► Indexes: (3)

▼ Attributes:

Conventions : COARDS
 title : 4x daily NMC reanalysis (1948)
 description : Data is from NMC initialized reanalysis (4x/day). These are the 0.9950 sigma level values.
 platform : Model
 references : <http://www.esrl.noaa.gov/psd/data/gridded/data.ncep.reanalysis.html>

Figure 28: The data structure of Xarray's Dataset.

18.6. Working with DataArrays

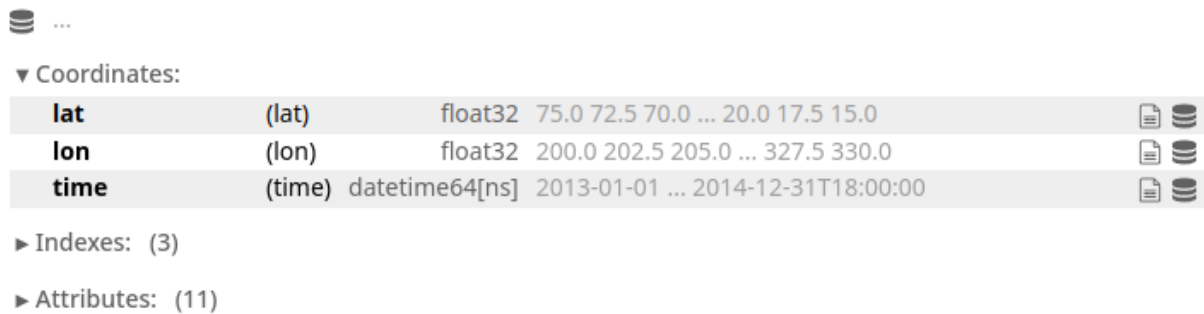
The heart of Xarray is the DataArray, which represents a single variable with all its associated coordinate and metadata information.

18.6.1. Accessing DataArrays from Datasets

You can extract individual DataArrays from a Dataset using dictionary-style notation or attribute access (Figure 29).

```
# Extract the air temperature DataArray using dictionary notation
temperature = ds["air"]
temperature
```

xarray.DataArray 'air' (time: 2920, lat: 25, lon: 53)



The figure shows the structure of an Xarray DataArray named 'air'. It displays the coordinates, indexes, and attributes. The coordinates are lat, lon, and time. The indexes are (3) and the attributes are (11).

Coordinates:			
lat	(lat)	float32	75.0 72.5 70.0 ... 20.0 17.5 15.0
lon	(lon)	float32	200.0 202.5 205.0 ... 327.5 330.0
time	(time)	datetime64[ns]	2013-01-01 ... 2014-12-31T18:00:00

► Indexes: (3)

► Attributes: (11)

Figure 29: The data structure of Xarray's DataArray.

Alternatively, you can use dot notation for cleaner syntax:

```
# Same result using attribute access
temperature = ds.air
temperature
```

Both approaches yield the same result, but dot notation is often more readable in complex analyses.

18.6.2. Exploring DataArray Components

Understanding the components of a DataArray helps you work more effectively with your data:

```
# Examine the actual data values (a NumPy array)
print("Data shape:", temperature.values.shape)
print("Data type:", temperature.values.dtype)
print("First few values:", temperature.values.flat[:5])
```

```
Data shape: (2920, 25, 53)
Data type: float64
First few values: [241.2 242.5 243.5 244. 244.1]
```

```
# Understand the dimension structure
print("Dimensions:", temperature.dims)
print("Dimension sizes:", temperature.sizes)
```

```
print("Dimensions:", temperature.dims)
print("Dimension sizes:", temperature.sizes)
```

```
# Explore the coordinate information
print("Coordinates:")
for name, coord in temperature.coords.items():
    print(f" {name}: {coord.values[:3]}... (showing first 3 values)")
```

```
Coordinates:
  lat: [75.  72.5 70.  ]... (showing first 3 values)
  lon: [200.  202.5 205.  ]... (showing first 3 values)
  time: ['2013-01-01T00:00:00.000000000' '2013-01-01T06:00:00.000000000'
        '2013-01-01T12:00:00.000000000']... (showing first 3 values)
```

```
# Examine metadata attributes
print("Attributes:")
for key, value in temperature.attrs.items():
    print(f" {key}: {value}")
```

```
Attributes:
  long_name: 4xDaily Air temperature at sigma level 995
  units: degK
  precision: 2
  GRIB_id: 11
  GRIB_name: TMP
  var_desc: Air temperature
  dataset: NMC Reanalysis
  level_desc: Surface
  statistic: Individual Obs
  parent_stat: Other
  actual_range: [185.16 322.1 ]
```

Why this matters:

- **Values:** Understanding data type and shape helps with memory management and numerical precision
- **Dimensions:** Knowing dimension order and names enables correct indexing and operations
- **Coordinates:** Coordinate values are crucial for spatial and temporal analysis
- **Attributes:** Metadata provides essential context for interpreting and documenting results

18.7. Intuitive Data Selection and Indexing

One of Xarray's greatest strengths is its intuitive approach to data selection. Instead of remembering array indices, you work directly with coordinate values.

18.7.1. Label-Based Selection

Label-based selection lets you choose data using actual coordinate values—dates, locations, measurement levels—rather than array positions:

```
# Select data for a specific date and location
point_data = temperature.sel(time="2013-01-01", lat=40.0, lon=260.0)
point_data
```

This approach is intuitive and self-documenting. Anyone reading your code immediately understands what data you're selecting.

18.7.2. Time Range Selection

Working with time series often requires selecting data within specific time ranges:

```
# Select all data for January 2013
january_data = temperature.sel(time=slice("2013-01-01", "2013-01-31"))
print(f"January 2013 data shape: {january_data.shape}")
print(f"Time range: {january_data.time.values[0]} to {january_data.time.values[-1]}")
```

Slice notation: The `slice()` function creates a range selector, similar to Python's standard slicing but using coordinate values instead of indices.

18.7.3. Nearest Neighbor Selection

Real-world coordinates often don't match exact grid points. Xarray can automatically find the nearest available data:

```
# Select data nearest to a location that might not be exactly on the grid
nearest_data = temperature.sel(lat=40.5, lon=255.7, method="nearest")
actual_coords = nearest_data.sel(time="2013-01-01")
print(f"Requested: lat=40.5, lon=255.7")
print(f"Actual: lat={actual_coords.lat.values}, lon={actual_coords.lon.values}")
```

This feature is invaluable when working with irregularly spaced data or when your analysis points don't exactly match measurement locations.

18.8. Performing Operations on Multi-Dimensional Data

Xarray's true power emerges when performing operations across multiple dimensions. The library handles dimension alignment, broadcasting, and coordinate management automatically.

18.8.1. Statistical Operations Across Dimensions

Computing statistics across specific dimensions is a common task in geospatial analysis:

```
# Calculate the temporal mean (average temperature at each location)
mean_temperature = temperature.mean(dim="time")
print(f"Original data shape: {temperature.shape}")
```

```
print(f"Time-averaged data shape: {mean_temperature.shape}")
print(
    f"Temperature range: {mean_temperature.min().values:.1f} to "
    f"{mean_temperature.max().values:.1f} K"
)
```

```
Original data shape: (2920, 25, 53)
Time-averaged data shape: (25, 53)
Temperature range: 249.1 to 301.6 K
```

What happened: By specifying `dim="time"`, we computed the average across all time steps, leaving us with a 2D array representing the long-term average temperature at each geographic location.

18.8.2. Computing Anomalies

Climate analysis often involves computing anomalies—deviations from long-term averages:

```
# Calculate temperature anomalies by subtracting the time mean from each time
step
anomalies = temperature - mean_temperature
print(f"Anomaly range: {anomalies.min().values:.1f} to "
      f"{anomalies.max().values:.1f} K")

# Find the location and time of the largest positive anomaly
max_anomaly = anomalies.max()
max_location = anomalies.where(anomalies == max_anomaly, drop=True)
print(f"Largest positive anomaly: {max_anomaly.values:.1f} K")
```

```
Anomaly range: -42.1 to 39.9 K
Largest positive anomaly: 39.9 K
```

Broadcasting magic: Xarray automatically handles the broadcasting between the 3D temperature array and the 2D mean temperature array, aligning coordinates correctly without any manual intervention.

18.8.3. Spatial Statistics

Computing statistics across spatial dimensions helps identify temporal patterns:

```
# Calculate area-weighted spatial mean for each time step
spatial_mean = temperature.mean(dim=["lat", "lon"])
print(f"Spatial mean temperature time series shape: {spatial_mean.shape}")

# Find the warmest and coldest time periods
warmest_date = spatial_mean.time[spatial_mean.argmax()]
coldest_date = spatial_mean.time[spatial_mean.argmin()]
```

```
print(f"Warmest period: {warmest_date.values}")
print(f"Coldest period: {coldest_date.values}")
```

```
Spatial mean temperature time series shape: (2920,)
Warmest period: 2013-08-12T00:00:00.000000000
Coldest period: 2014-02-12T12:00:00.000000000
```

18.9. Data Visualization with Xarray

Xarray's integration with Matplotlib makes creating publication-quality visualizations straightforward, with automatic handling of coordinates and metadata.

18.9.1. Plotting 2D Spatial Data

Visualizing spatial patterns is essential for understanding geospatial data ([Figure 30](#)):

```
# Create a map of long-term average temperature
fig, ax = plt.subplots(figsize=(12, 6))
mean_temperature.plot(ax=ax, cmap="RdYlBu_r", add_colorbar=True)
plt.title("Long-term Average Air Temperature", fontsize=14, fontweight="bold")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.tight_layout()
plt.show()
```

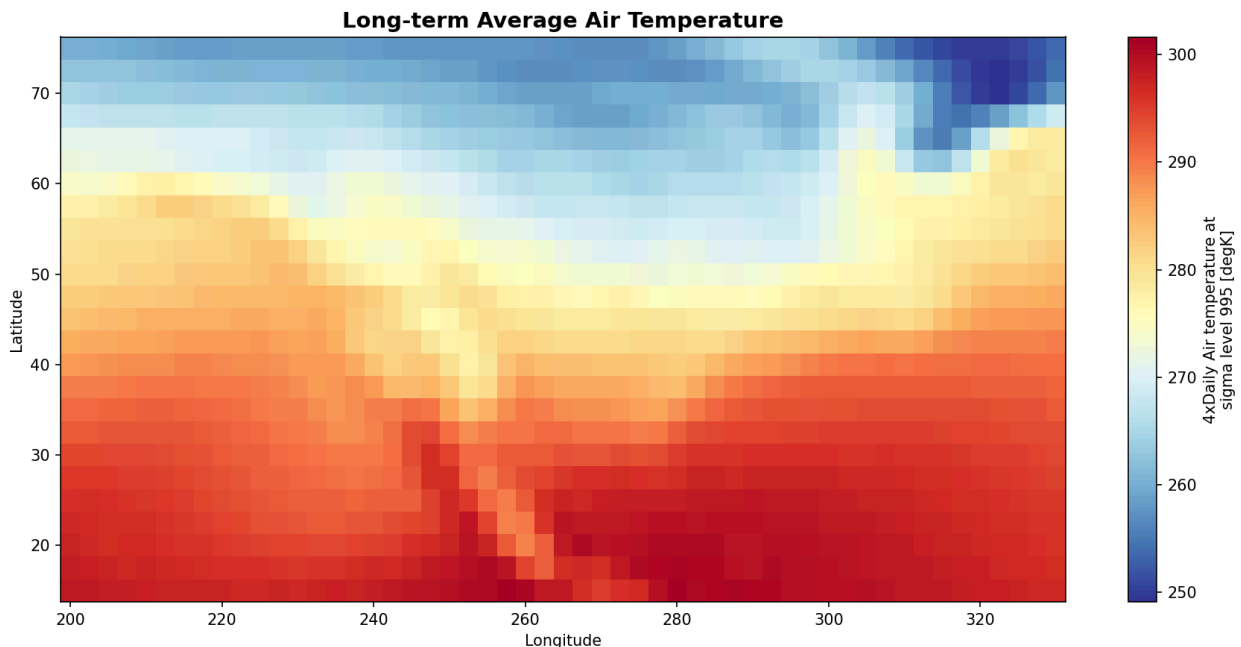


Figure 30: The long-term average temperature of the dataset.

Automatic features: Notice how Xarray automatically:

- Uses coordinate values for axis labels
- Adds appropriate axis titles
- Creates a colorbar with proper units
- Handles map projection basics

18.9.2. Customizing Spatial Plots

You can customize visualizations to better communicate your findings. For example, you can specify the color map, add labels, and set the figure size (see [Figure 31](#)).

```
# Create a more customized visualization
fig, ax = plt.subplots(figsize=(12, 6))
plot = mean_temperature.plot(
    ax=ax,
    cmap="RdYlBu_r",
    levels=20, # Number of contour levels
    add_colorbar=True,
    cbar_kwargs={"label": "Temperature (K)", "shrink": 0.8, "pad": 0.02},
)
plt.title("Mean Air Temperature (2013)", fontsize=16, fontweight="bold")
plt.xlabel("Longitude (°E)", fontsize=12)
plt.ylabel("Latitude (°N)", fontsize=12)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

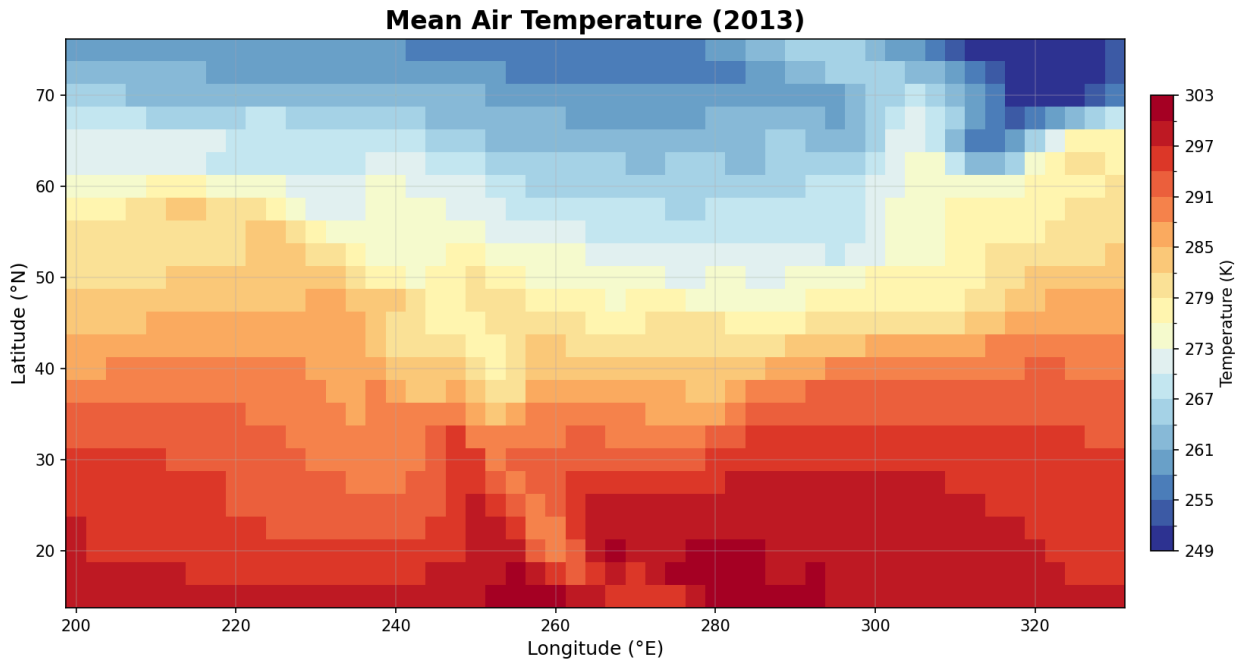


Figure 31: A custom plot of the mean temperature of the dataset.

18.9.3. Time Series Visualization

Plotting temporal data at specific locations reveals patterns over time. For example, we can plot the temperature time series at a specific location (Figure 32):

```
# Select and plot time series for a specific location
location_ts = temperature.sel(lat=40.0, lon=260.0)

fig, ax = plt.subplots(figsize=(12, 6))
location_ts.plot(ax=ax, linewidth=1.5, color="darkblue")
plt.title("Temperature Time Series at 40°N, 260°E", fontsize=14,
fontweight="bold")
plt.xlabel("Time")
plt.ylabel("Temperature (K)")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

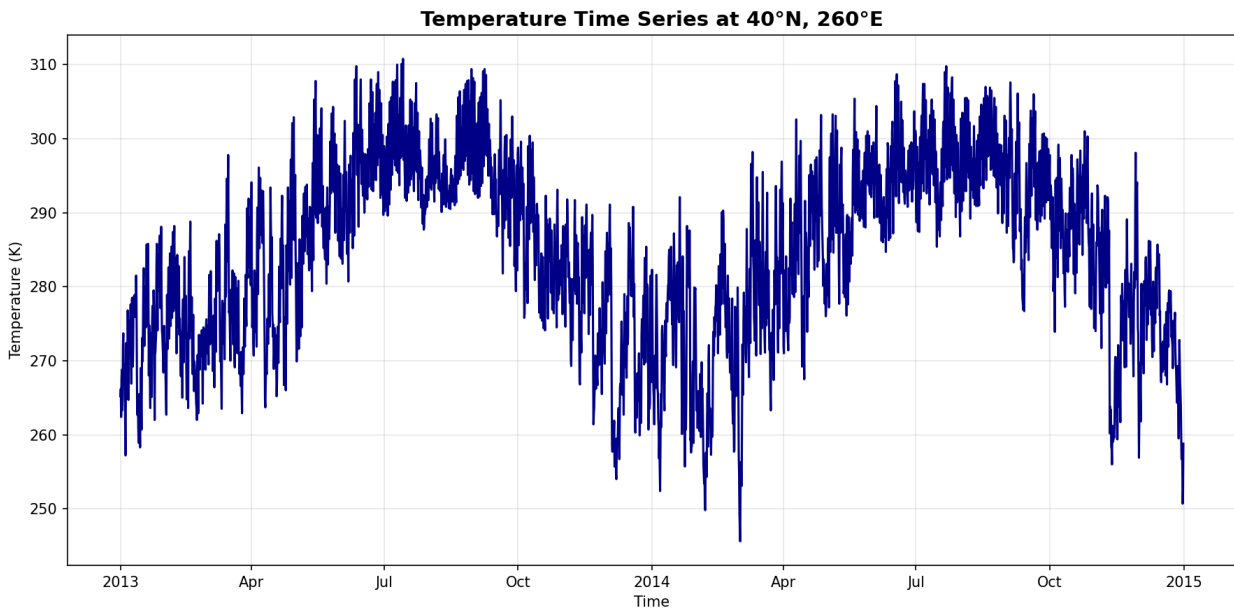


Figure 32: A time series plot of the temperature at a specific location.

Automatic time handling: Xarray correctly interprets and displays time coordinates, handling date formatting and axis spacing automatically.

18.10. Working with Datasets: Multiple Variables

Real-world analysis often involves multiple related variables. Datasets allow you to work with these variables coherently while maintaining their relationships.

18.10.1. Exploring Dataset Structure

Understanding your dataset's structure is the first step in any analysis:

```
# Examine all variables in the dataset
print("Data variables in the dataset:")
for var_name, var_info in ds.data_vars.items():
    print(f" {var_name}: {var_info.dims}, shape {var_info.shape}")

print(f"\nShared coordinates: {list(ds.coords.keys())}")
print(f"Global attributes: {len(ds.attrs)} metadata items")
```

```
Data variables in the dataset:
  air: ('time', 'lat', 'lon'), shape (2920, 25, 53)

Shared coordinates: ['lat', 'lon', 'time']
Global attributes: 5 metadata items
```

18.10.2. Dataset-Level Operations

Operations can be applied to entire datasets, affecting all variables consistently:

```
# Calculate temporal statistics for all variables in the dataset
dataset_means = ds.mean(dim="time")
dataset_means
```

This approach ensures consistent processing across all variables while maintaining their coordinate relationships.

18.11. The Power of Label-Based Operations

Xarray's label-based approach becomes particularly powerful when compared to working with raw NumPy arrays.

18.11.1. The NumPy Approach: Index-Based Selection

Here's how you might work with the same data using NumPy arrays:

```
# Extract raw arrays and coordinates
lat_values = ds.air.lat.values
lon_values = ds.air.lon.values
temp_values = ds.air.values

print(f"Data shape: {temp_values.shape}")
print("To plot the first time step, you need to remember:")
print("- Time is dimension 0")
```

```
print("- Latitude is dimension 1")
print("- Longitude is dimension 2")
```

```
# Plot using NumPy approach - requires careful index management
fig, ax = plt.subplots(figsize=(12, 6))
im = ax.pcolormesh(lon_values, lat_values, temp_values[0, :, :], cmap="RdYlBu_r")
plt.colorbar(im, ax=ax, label="Temperature (K)")
plt.title("First Time Step (NumPy approach)")
plt.xlabel("Longitude")
plt.ylabel("Latitude")
plt.show()
```

18.11.2. The Xarray Approach: Label-Based Selection

Compare this with Xarray's intuitive approach:

```
# Same result with Xarray - much more readable and less error-prone
ds.air.isel(time=0).plot(figsize=(12, 6), cmap="RdYlBu_r")
plt.title("First Time Step (Xarray approach)")
plt.show()
```

Or even better, using meaningful date selection, such as “2013-01-01T00:00:00”. The result is a DataArray with a single time step ([Figure 33](#)).

```
# Select by actual date rather than array index
ds.air.sel(time="2013-01-01T00:00:00").plot(figsize=(12, 6), cmap="RdYlBu_r")
plt.title("Temperature on January 1, 2013")
plt.show()
```

Key advantages of the Xarray approach:

- **Self-documenting:** Code clearly shows what data is being selected
- **Error-resistant:** No need to remember dimension order or calculate indices
- **Flexible:** Easy to change selections without rewriting indexing logic
- **Maintainable:** Code remains readable even after months away from the project

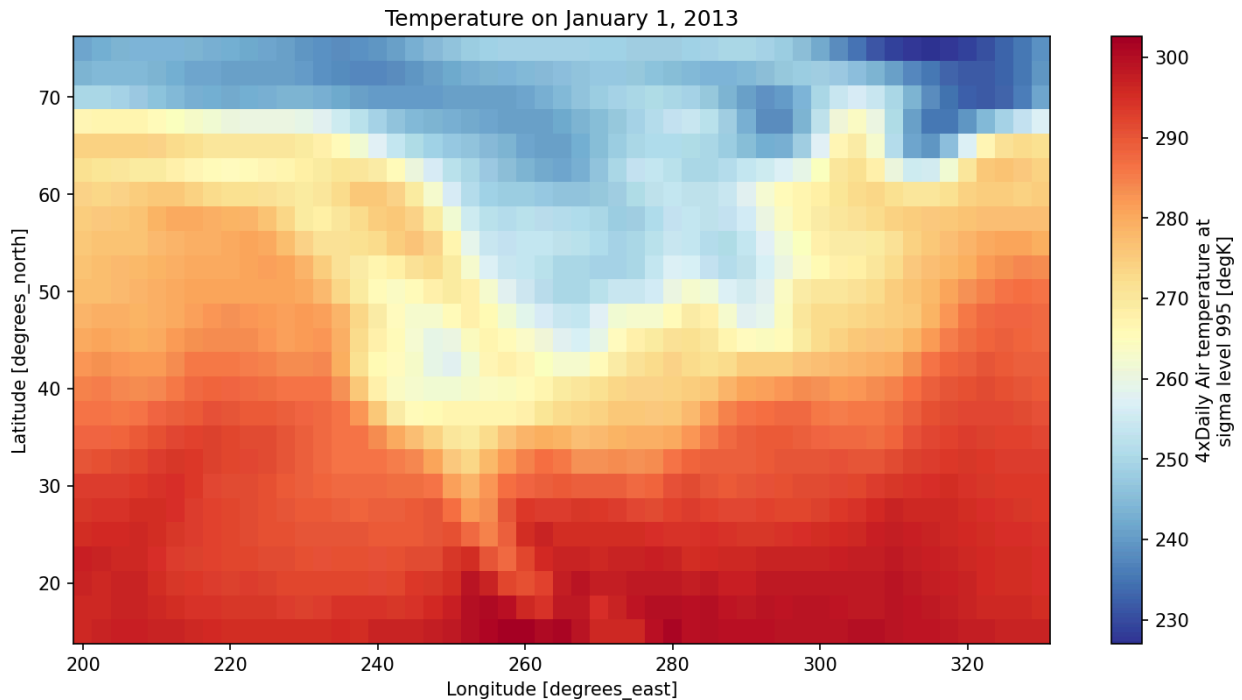


Figure 33: The temperature on January 1, 2013.

18.12. Advanced Indexing Techniques

Xarray provides flexible indexing methods that accommodate different data selection needs.

18.12.1. Position-Based vs. Label-Based Indexing

Understanding when to use each approach helps you work more efficiently. The `isel()` method is useful for systematic sampling, while `sel()` is ideal for specific values or date ranges.

```
# Position-based indexing using isel() - useful for systematic sampling
first_last_times = ds.air.isel(time=[0, -1]) # First and last time steps
print(f"Selected time steps: {first_last_times.time.values}")

# Label-based indexing using sel() - useful for specific values
specific_months = ds.air.sel(time=slice("2013-05", "2013-07"))
print(f"May-July 2013 contains {len(specific_months.time)} time steps")
```

18.12.2. Boolean Indexing and Conditional Selection

Select data based on conditions to focus analysis on interesting subsets:

```
# Find locations where average temperature exceeds a threshold
warm_locations = mean_temperature.where(mean_temperature > 280) # 280 K ≈ 7°C
```

```
warm_count = warm_locations.count()
print(f"Number of grid points with mean temperature > 280 K:
{warm_count.values}")

# Find time periods when spatial average temperature was unusually high
temp_threshold = spatial_mean.quantile(0.9) # 90th percentile
warm_periods = spatial_mean.where(spatial_mean > temp_threshold, drop=True)
print(f"Number of exceptionally warm time periods: {len(warm_periods)}")
```

In the example above, we selected locations where the mean temperature was above 280 K and counted the number of grid points that met this criterion. We also identified locations where the mean temperature was unusually high. These kinds of operations are common in geospatial analysis, and Xarray makes them straightforward.

18.13. High-Level Computational Operations

Xarray provides powerful high-level operations that make complex analyses straightforward and readable, such as `groupby`, `resample`, `rolling`, and `weighted`.

18.13.1. GroupBy Operations for Temporal Analysis

GroupBy operations enable analysis across meaningful time periods. For example, we can calculate the seasonal climatology of the temperature dataset by simply grouping by the season using the `groupby` method:

```
# Calculate seasonal climatology
seasonal_means = ds.air.groupby("time.season").mean()
print("Seasonal temperature patterns:")
seasonal_means
```

The resulting `seasonal_means` is a DataArray with the seasonal mean temperature for each season (DJF, MAM, JJA, SON). We can plot it using the `plot` method (see [Figure 34](#)):

```
# Visualize seasonal patterns
fig, axes = plt.subplots(2, 2, figsize=(12, 6))
seasons = ["DJF", "MAM", "JJA", "SON"]
season_names = ["Winter", "Spring", "Summer", "Fall"]

for i, (season, name) in enumerate(zip(seasons, season_names)):
    ax = axes[i // 2, i % 2]
    seasonal_means.sel(season=season).plot(ax=ax, cmap="RdYlBu_r",
add_colorbar=False)
    ax.set_title(f"{name} ({season})")
    ax.set_xlabel("Longitude")
    ax.set_ylabel("Latitude")
```

```
plt.tight_layout()
plt.show()
```

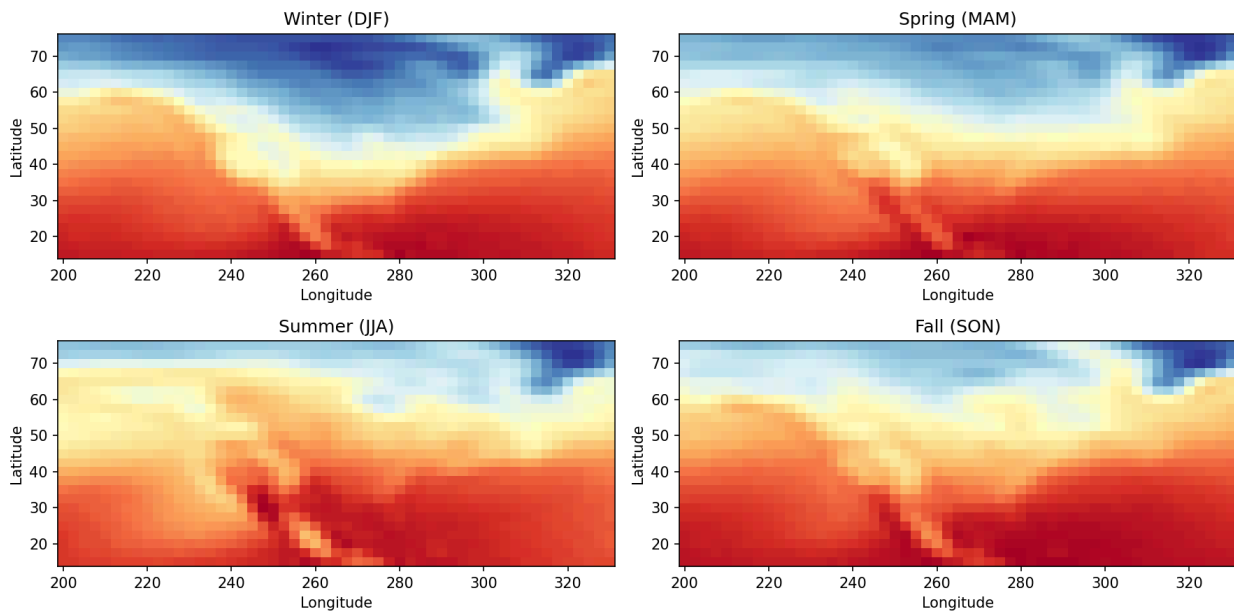


Figure 34: The seasonal mean temperature of the dataset.

18.13.2. Rolling Window Operations

Smoothing operations help identify trends and reduce noise. For example, we can smooth the temperature time series at a specific location using a rolling window ([Figure 35](#)):

```
# Create a smoothed time series using a rolling window
location_data = temperature.sel(lat=40.0, lon=260.0)

fig, ax = plt.subplots(figsize=(12, 6))

# Plot original data
location_data.plot(ax=ax, alpha=0.5, label="Original", color="lightblue")

# Plot smoothed data using a 30-day rolling window
smoothed_data = location_data.rolling(time=30, center=True).mean()
smoothed_data.plot(ax=ax, label="30-day smoothed", color="darkblue", linewidth=2)

plt.title("Temperature Time Series: Original vs Smoothed")
plt.xlabel("Time")
plt.ylabel("Temperature (K)")
plt.legend()
plt.grid(True, alpha=0.3)
```

```
plt.tight_layout()
plt.show()
```

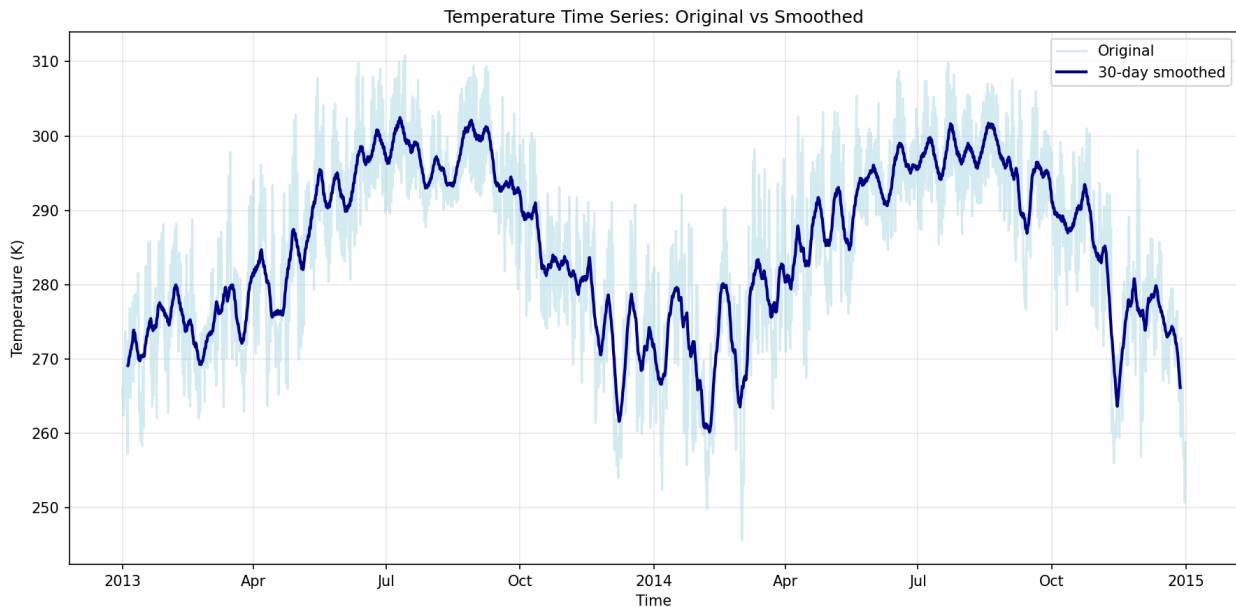


Figure 35: A rolling window operation on the temperature dataset.

18.13.3. Weighted Operations

In geospatial analysis, weighting operations by factors like area or population is often necessary. The example below illustrates the difference between a simple mean (treating all grid cells equally) and an area-weighted mean (which accounts for the fact that grid cells near the poles represent less physical area than those near the equator). The resulting plot is shown in [Figure 36](#).

```
# Create simple area weights (this is a simplified example)
# In practice, you would use proper latitude-based area weighting
lat_weights = np.cos(np.radians(ds.air.lat))
area_weighted_mean = ds.air.weighted(lat_weights).mean(dim=["lat", "lon"])

# Compare simple vs area-weighted spatial averages
fig, ax = plt.subplots(figsize=(12, 6))
spatial_mean.plot(ax=ax, label="Simple average", alpha=0.7)
area_weighted_mean.plot(ax=ax, label="Area-weighted average", linewidth=2)
plt.title("Spatial Temperature Averages: Simple vs Area-Weighted")
plt.xlabel("Time")
plt.ylabel("Temperature (K)")
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

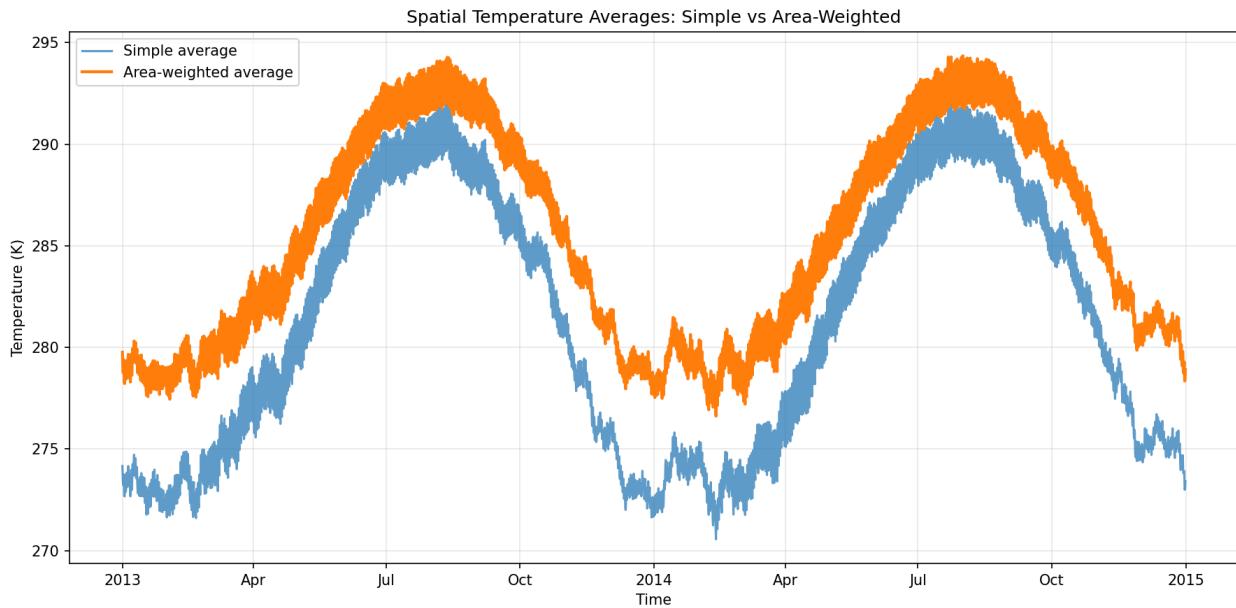


Figure 36: The weighted mean temperature of the dataset.

18.14. Data Input and Output

Scientific reproducibility requires reliable data storage and sharing. Xarray excels at working with standard scientific data formats.

18.14.1. Understanding NetCDF Format

NetCDF (Network Common Data Form) is the gold standard for scientific data storage because it:

- Preserves all metadata and coordinate information
- Supports efficient partial reading of large datasets
- Is platform-independent and widely supported
- Includes built-in data compression and chunking

18.14.2. Writing Data to NetCDF

Save your processed data for future use or sharing:

```
# Prepare data for saving (ensure proper data types)
output_ds = ds.copy()
output_ds["air"] = output_ds["air"].astype("float32") # Reduce file size

# Add processing metadata
output_ds.attrs["processing_date"] = str(np.datetime64("now"))
output_ds.attrs["created_by"] = "GIS Pro"

# Save to NetCDF file
```

```
output_ds.to_netcdf("processed_air_temperature.nc")
print("Dataset saved to processed_air_temperature.nc")
```

18.14.3. Reading Data from NetCDF

Load saved data, preserving all metadata and coordinate information:

```
# Load the saved dataset
reloaded_ds = xr.open_dataset("processed_air_temperature.nc")
print("Successfully reloaded dataset:")
print(f"Variables: {list(reloaded_ds.data_vars.keys())}")
print(f"Processing date: {reloaded_ds.attrs.get('processing_date', 'Not specified')}")
print(f>Data matches original: {reloaded_ds.air.equals(ds.air.astype('float32'))}")
```

Best practices for data storage:

- Use descriptive filenames and include dates
- Add meaningful metadata attributes
- Choose appropriate data types to balance precision and file size
- Document your processing steps in the metadata

18.15. Key Takeaways

This chapter introduced the fundamental concepts and operations of Xarray for multi-dimensional data analysis. The key principles to remember:

Intuitive Data Model: Xarray's labeled dimensions and coordinates make multi-dimensional data analysis intuitive and self-documenting, eliminating the error-prone index management required with raw NumPy arrays.

Automatic Alignment and Broadcasting: Xarray handles coordinate alignment and array broadcasting automatically, letting you focus on the analysis rather than the mechanics of array manipulation.

Integrated Metadata: Attributes and coordinate information travel with your data through all operations, ensuring reproducible and well-documented analyses.

Seamless Visualization: Built-in plotting methods create publication-quality visualizations with minimal code, automatically handling coordinate labeling and formatting.

Scientific Data Integration: Native support for formats like NetCDF ensures seamless integration with the broader scientific computing ecosystem.

Scalable Operations: High-level operations like groupby, rolling windows, and weighted calculations make complex analyses accessible while maintaining computational efficiency.

Xarray bridges the gap between the complexity of multi-dimensional scientific data and the simplicity needed for effective analysis. By providing an intuitive interface that preserves the richness of scientific datasets, Xarray enables researchers to focus on scientific insights rather than data manipulation mechanics.

18.16. Further Reading

Xarray provides a wealth of resources for learning more about the library and its capabilities. Here are some of the most useful resources:

- Xarray documentation: <https://docs.xarray.dev>
- Xarray tutorial: <https://tutorial.xarray.dev>
- Xarray gallery: <https://docs.xarray.dev/en/stable/gallery.html>

18.17. Exercises

The following exercises will help you practice essential Xarray concepts using different datasets and analysis scenarios.

18.17.1. Exercise 1: Exploring a New Dataset

This exercise introduces you to a different climate dataset and helps you practice basic data exploration techniques.

1. Load the Xarray tutorial dataset `rasm` (Regional Arctic System Model data).
2. Inspect the `Dataset` object and list all the variables and dimensions.
3. Select the `Tair` variable (air temperature) and examine its structure.
4. Print the attributes, dimensions, and coordinates of `Tair` to understand the data better.

18.17.2. Exercise 2: Data Selection and Indexing

Practice different methods of selecting subsets from multi-dimensional data.

1. From the `Tair` data, select a subset for the date `1980-07-01` and latitude `70.0`.
2. Create a time slice for the entire latitude range between January and March of 1980.
3. Plot the selected time slice as a line plot to visualize the temporal pattern.

18.17.3. Exercise 3: Performing Arithmetic Operations

Learn to compute derived quantities and anomalies from climate data.

1. Compute the temporal mean of the `Tair` data over the `time` dimension.
2. Subtract the computed mean from the original `Tair` dataset to calculate temperature anomalies.
3. Create two plots: one showing the mean temperature pattern and another showing an example anomaly pattern.

18.17.4. Exercise 4: GroupBy and Temporal Analysis

Explore temporal patterns using Xarray's powerful groupby functionality.

1. Use `groupby` to calculate the seasonal mean temperature (`Tair`) across all years.
2. Use `resample` to calculate the monthly mean temperature for 1980.
3. Create visualizations showing both the seasonal patterns and the monthly progression for 1980.

18.17.5. Exercise 5: Data Storage and Retrieval

Practice saving and loading processed datasets while preserving metadata.

1. Select the temperature anomalies calculated in Exercise 3.
2. Convert the `Tair` variable to `float32` data type to optimize file size.
3. Add metadata attributes describing your processing steps.
4. Write the anomalies data to a new NetCDF file named `tair_anomalies.nc`.
5. Load the data back from the file and verify that it matches your original calculations.

Chapter 19. Raster Analysis with Rioxarray

19.1. Introduction

Imagine you're working with satellite imagery that covers vast geographic areas, or analyzing elevation data that represents terrain across an entire country. These datasets are fundamentally different from the vector data we explored in previous chapters—instead of discrete points, lines, and polygons, we're working with continuous grids of values that cover space like a digital blanket. This is where [rioxarray](https://corteva.github.io/rioxarray)⁵⁷ becomes an invaluable tool for geospatial analysis.

Rioxarray bridges two powerful Python ecosystems: the multi-dimensional array capabilities of Xarray and the geospatial data handling prowess of [Rasterio](https://rasterio.readthedocs.io)⁵⁸. Think of it as Xarray that has learned to speak the language of geospatial data—it understands coordinate reference systems, spatial transformations, and all the complexities that come with georeferenced raster data.

Key advantages of rioxarray:

- **Geospatial awareness:** Full integration of coordinate reference systems, spatial transformations, and georeferencing information
- **Xarray integration:** All the power of labeled, multi-dimensional arrays with automatic alignment and broadcasting
- **Format flexibility:** Native support for common geospatial formats like GeoTIFF, NetCDF, and more
- **Efficient operations:** Optimized reading and writing of large raster datasets
- **Spatial operations:** Built-in methods for reprojection, clipping, resampling, and masking
- **Metadata preservation:** Automatic handling of spatial metadata through all operations

19.2. Learning Objectives

By the end of this chapter, you should be able to:

- Understand how rioxarray extends Xarray for comprehensive geospatial raster data handling
- Load and inspect georeferenced raster datasets with full spatial metadata awareness
- Perform essential geospatial operations including reprojection, clipping, and spatial subsetting
- Work confidently with coordinate reference systems and spatial transformations in raster data
- Apply basic raster analysis techniques such as band mathematics and vegetation index calculations
- Export processed raster data while preserving spatial metadata and georeferencing information
- Visualize raster data effectively using matplotlib with proper spatial context

19.3. Setting Up Your Rioxarray Environment

Before diving into raster analysis, let's ensure you have the necessary tools installed and properly configured.

19.3.1. Installing Required Libraries

Rioxarray depends on several geospatial libraries, but modern package managers handle these dependencies automatically. Uncomment the following code to install the required libraries:

⁵⁷<https://corteva.github.io/rioxarray>

⁵⁸<https://rasterio.readthedocs.io>

```
# %pip install rioxarray pygis
```

19.3.2. Importing Libraries and Configuration

To start, we need to import the necessary libraries and configure Xarray for better display.

```
import rioxarray
import numpy as np
import xarray as xr
import matplotlib.pyplot as plt

# Configure Xarray for better display
xr.set_options(keep_attrs=True, display_expand_data=False)

# Configure matplotlib for high-quality plots
plt.rcParams["figure.dpi"] = 150
```

Why these imports matter:

- **rioxarray**: Provides the geospatial extensions to Xarray
- **numpy**: Essential for numerical operations on array data
- **xarray**: The foundation for multi-dimensional labeled arrays
- **matplotlib**: For creating visualizations and maps

19.4. Loading and Exploring Georeferenced Raster Data

The foundation of raster analysis is understanding how to load and interpret geospatial raster datasets. Unlike regular arrays, georeferenced rasters carry crucial spatial information that determines where each pixel exists in real-world coordinates.

19.4.1. Understanding Raster Data Structure

Before loading data, it's helpful to understand what makes raster data "geospatial." A georeferenced raster contains:

- **Pixel values**: The actual measurements or classifications stored in each grid cell
- **Spatial dimensions**: Usually represented as rows and columns (y and x in geographic terms)
- **Coordinate reference system (CRS)**: Defines how the pixel coordinates relate to Earth's surface
- **Affine transformation**: Mathematical relationship between pixel coordinates and geographic coordinates
- **Metadata**: Information about the data source, collection methods, and processing history

19.4.2. Loading a Real Satellite Image

Let's start with a practical example using a Landsat satellite image:

```
# Load a Landsat satellite image covering Las Vegas area
url = "https://github.com/opengeos/datasets/releases/download/raster/LC09_039035_20240708_90m.tif"
data = rioarray.open_rasterio(url)
print(f"Successfully loaded raster with shape: {data.shape}")
data
```

Understanding the loaded data:

The `open_rasterio()` function creates an Xarray DataArray with additional geospatial capabilities. Notice how the output shows not just the array dimensions, but also the coordinate reference system and spatial extent.

19.4.3. Exploring the Dataset Structure

Let's examine the key components of our raster dataset:

```
# Examine the basic structure
print("Dataset dimensions:", data.dims)
print("Dataset shape:", data.shape)
print("Data type:", data.dtype)
```

```
Dataset dimensions: ('band', 'y', 'x')
Dataset shape: (7, 2563, 2485)
Data type: float32
```

```
# Explore the coordinate information
print("Coordinates:")
for name, coord in data.coords.items():
    print(f" {name}: {coord.values.min()} to {coord.values.max()}")
```

```
Coordinates:
  band: 1 to 7
    x: 582435.0 to 805995.0
    y: 3874995.0 to 4105575.0
 spatial_ref: 0 to 0
```

```
# Check metadata attributes
print("\nKey attributes:")
for key, value in data.attrs.items():
    if key in ["long_name", "grid_mapping", "AREA_OR_POINT"]:
        print(f" {key}: {value}")
```

Key attributes:

AREA_OR_POINT: Area

long_name: ('SR_B1', 'SR_B2', 'SR_B3', 'SR_B4', 'SR_B5', 'SR_B6', 'SR_B7')

Interpreting the structure:

- **Band dimension:** Landsat images typically have multiple spectral bands (Blue, Green, Red, Near-Infrared, etc.)
- **Y and X dimensions:** Represent the spatial grid, with Y typically running north-south and X running east-west
- **Coordinate values:** Show the actual geographic locations (in the dataset's coordinate system)

19.4.4. Accessing Spatial Reference Information

The spatial reference information is crucial for any geospatial analysis:

```
# Check the coordinate reference system
print(f"Coordinate Reference System: {data.rio.crs}")
```

Coordinate Reference System: EPSG:32611

```
# Examine the affine transformation
print("\nAffine transformation:")
transform = data.rio.transform()
print(transform)
print(f"Pixel size: {abs(transform[0]):.1f} x {abs(transform[4]):.1f} meters")
```

```
Affine transformation:
| 90.00, 0.00, 582390.00|
| 0.00,-90.00, 4105620.00|
| 0.00, 0.00, 1.00|
Pixel size: 90.0 x 90.0 meters
```

Understanding the CRS and transform:

- **CRS:** Defines how coordinates relate to locations on Earth's surface
- **Affine transformation:** Converts pixel row/column indices to geographic coordinates
- **Pixel size:** Determines the spatial resolution of your data

19.4.5. Setting CRS When Missing

Sometimes raster data may lack proper CRS information. You can assign it manually:

```
# Example: Set CRS if missing or incorrect (only run if needed)
# data = data.rio.write_crs("EPSG:32611", inplace=True)
print(f"Current CRS is properly set: {data.rio.crs}")
```

This step is crucial because many spatial operations require known coordinate systems.

19.5. Fundamental Geospatial Operations

Rioarray provides essential tools for manipulating raster data spatially. These operations are the building blocks of most raster analysis workflows.

19.5.1. Coordinate System Transformations

Different projects often require data in different coordinate systems, making reprojection one of the most fundamental operations in geospatial analysis. Coordinate reference systems (CRS) define how coordinates on a map relate to locations on the Earth's surface, and choosing the right CRS can significantly impact your analysis results.

When you reproject raster data, several things happen automatically:

- Pixel values are resampled to fit the new grid structure
- The spatial extent changes to match the new coordinate system
- Pixel sizes may change, affecting the spatial resolution
- The grid alignment shifts to match the new projection's geometry

Rioarray handles all these transformations automatically while preserving the data's geospatial integrity. The `rio.reproject()` method ensures your data maintains its spatial accuracy and statistical properties.

```
# Reproject from UTM to Geographic coordinates (WGS84)
print(f"Original CRS: {data.rio.crs}")
data_geographic = data.rio.reproject("EPSG:4326")
print(f"Reprojected CRS: {data_geographic.rio.crs}")
print(f"New coordinate ranges:")
print(
    f"    Longitude: {data_geographic.x.min().values:.3f} to "
    f"{data_geographic.x.max().values:.3f}"
)
print(
    f"    Latitude: {data_geographic.y.min().values:.3f} to "
    f"{data_geographic.y.max().values:.3f}"
)
```

```
Original CRS: EPSG:32611
Reprojected CRS: EPSG:4326
New coordinate ranges:
    Longitude: -116.097 to -113.559
    Latitude: 34.972 to 37.093
```

19.5.2. Spatial Subsetting with Bounding Boxes

Often you need to focus on a specific geographic area. Bounding box clipping is the most straightforward approach:

```
# Define a bounding box around central Las Vegas
# (longitude_min, latitude_min, longitude_max, latitude_max)
vegas_bbox = [-115.391, 35.982, -114.988, 36.425]

print("Clipping raster to Las Vegas area...")
clipped_data = data_geographic.rio.clip_box(*vegas_bbox)
print(f"Original size: {data_geographic.shape}")
print(f"Clipped size: {clipped_data.shape}")
```

```
Clipping raster to Las Vegas area...
Original size: (7, 2351, 2812)
Clipped size: (7, 492, 447)
```

19.5.3. Vector-Based Clipping

For more precise spatial subsetting, you can use vector geometries to clip raster data:

```
import geopandas as gpd

# Load a vector boundary for more precise clipping
boundary_url = "https://github.com/opengeos/datasets/releases/download/places/
las_vegas_bounds_utm.geojson"
boundary = gpd.read_file(boundary_url)

print(f"Boundary CRS: {boundary.crs}")
print(f"Raster CRS: {data.rio.crs}")

# Clip the raster using the vector boundary
clipped_by_vector = data.rio.clip(boundary.geometry, boundary.crs)
print(f"Vector-clipped size: {clipped_by_vector.shape}")
```

```
Boundary CRS: EPSG:32611
Raster CRS: EPSG:32611
Vector-clipped size: (7, 522, 514)
```

19.6. Working with Spatial Dimensions and Resolution

Understanding and manipulating spatial dimensions is crucial for raster analysis, especially when integrating datasets or optimizing processing performance.

19.6.1. Understanding Spatial Resolution

Spatial resolution determines the level of detail in your raster data:

```
# Calculate current spatial resolution
transform = data.rio.transform()
x_resolution = abs(transform[0])
y_resolution = abs(transform[4])

print(f"Current spatial resolution:")
print(f"  X (East-West): {x_resolution:.1f} meters")
print(f"  Y (North-South): {y_resolution:.1f} meters")

# Calculate total area covered
width_km = (data.x.max() - data.x.min()).values / 1000
height_km = (data.y.max() - data.y.min()).values / 1000
print(f"\nSpatial extent:")
print(f"  Width: {width_km:.1f} km")
print(f"  Height: {height_km:.1f} km")
```

```
Current spatial resolution:
  X (East-West): 90.0 meters
  Y (North-South): 90.0 meters
```

```
Spatial extent:
  Width: 223.6 km
  Height: 230.6 km
```

19.6.2. Resampling to Different Resolutions

Sometimes you need to change the spatial resolution of your data. For example, you might want to resample an imagery dataset to a lower resolution for faster processing or to match the resolution of another dataset. The example below shows how to resample a 90-meter resolution dataset to 1km resolution:

```
# Resample to 1km resolution for regional analysis
print("Resampling to 1km resolution...")
resampled_data = data.rio.reproject(
    data.rio.crs, resolution=(1000, 1000) # Keep the same CRS # 1km x 1km
    pixels
)

print(f"Original shape: {data.shape}")
print(f"Resampled shape: {resampled_data.shape}")

# Calculate the reduction in data size
original_pixels = np.prod(data.shape)
```

```
resampled_pixels = np.prod(resampled_data.shape)
reduction_factor = original_pixels / resampled_pixels
print(f>Data reduction factor: {reduction_factor:.1f}x")
```

```
Resampling to 1km resolution...
Original shape: (7, 2563, 2485)
Resampled shape: (7, 231, 224)
Data reduction factor: 123.1x
```

19.6.3. Spatial Subsetting by Coordinates

You can also extract spatial subsets using coordinate ranges:

```
# Extract a subset using coordinate ranges (in geographic coordinates)
lon_range = (-115.391, -114.988)
lat_range = (35.982, 36.425)

print("Selecting subset by coordinate ranges...")
subset = data_geographic.sel(
    x=slice(*lon_range),
    y=slice(lat_range[1], lat_range[0]), # Note: y coordinates often decrease
)

print(f"Subset shape: {subset.shape}")
print(f"Longitude range: {subset.x.min().values:.3f} to {subset.x.max().values:.3f}")
print(f"Latitude range: {subset.y.min().values:.3f} to {subset.y.max().values:.3f}")
```

```
Selecting subset by coordinate ranges...
Subset shape: (7, 491, 447)
Longitude range: -115.391 to -114.988
Latitude range: 35.983 to 36.425
```

19.7. Visualizing Geospatial Raster Data

Effective visualization is crucial for understanding raster data and communicating results. Rioxarray integrates seamlessly with matplotlib for creating publication-quality maps.

19.7.1. Creating True-Color Composite Images

Satellite imagery often contains multiple spectral bands. Creating RGB composites helps visualize the data in familiar ways. Below is an example of a true-color composite of the Landsat 9 image of the Las Vegas area ([Figure 37](#)).

```

# Create a true-color (RGB) composite using bands 4, 3, 2 (Red, Green, Blue)
fig, ax = plt.subplots(figsize=(6, 6))

# Select RGB bands and create composite
rgb_bands = data_geographic.sel(band=[4, 3, 2])
rgb_bands.plot.imshow(
    ax=ax,
    vmin=0,
    vmax=0.3, # Adjust based on data range
)
# ax.set_aspect('equal')
plt.title(
    "Landsat True-Color Composite - Las Vegas Area", fontsize=14,
    fontweight="bold"
)
plt.xlabel("Longitude (°)", fontsize=12)
plt.ylabel("Latitude (°)", fontsize=12)
plt.tight_layout()
plt.show()

```

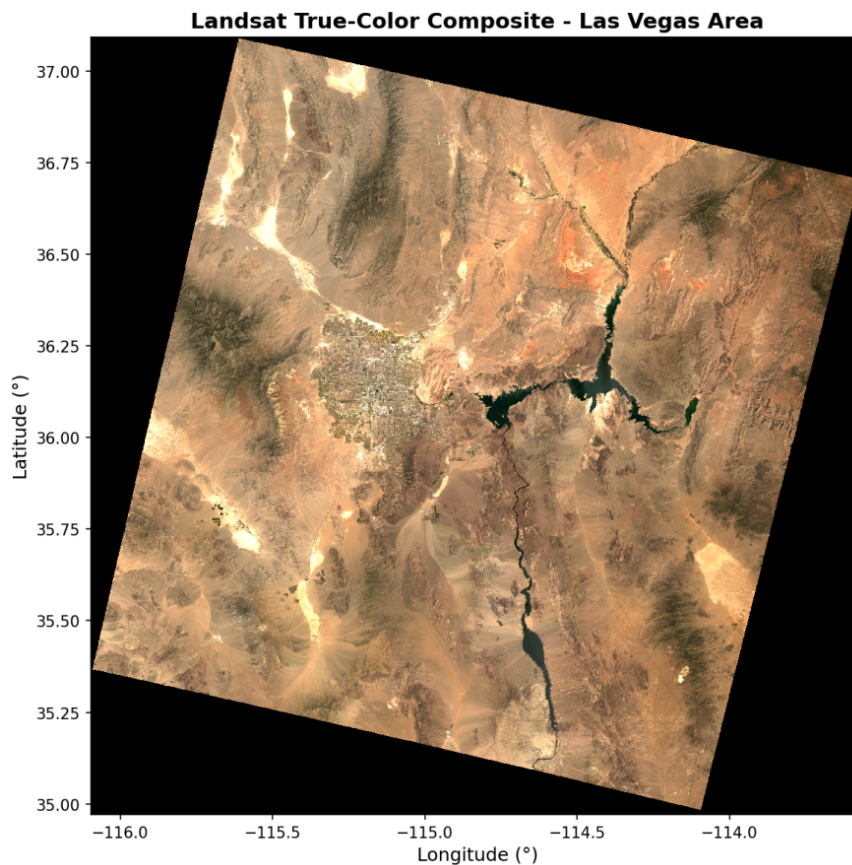


Figure 37: A true-color composite of the Landsat 9 image of the Las Vegas area.

19.7.2. Visualizing Individual Bands

Sometimes you need to examine individual spectral bands. For example, the near-infrared band (band 5) is useful for vegetation analysis ([Figure 38](#)).

```
# Plot a single band with proper spatial context
fig, ax = plt.subplots(figsize=(8, 6))

# Select near-infrared band (band 5) which is useful for vegetation analysis
nir_band = data_geographic.sel(band=5)
im = nir_band.plot.imshow(
    ax=ax,
    cmap="RdYlGn", # Red-Yellow-Green colormap
    vmin=0,
    vmax=0.5,
    add_colorbar=True,
    cbar_kwargs={"label": "Near-Infrared Reflectance", "shrink": 0.8},
)

plt.title("Near-Infrared Band - Vegetation Analysis", fontsize=14,
          fontweight="bold")
plt.xlabel("Longitude (°)", fontsize=12)
plt.ylabel("Latitude (°)", fontsize=12)
plt.tight_layout()
plt.show()
```

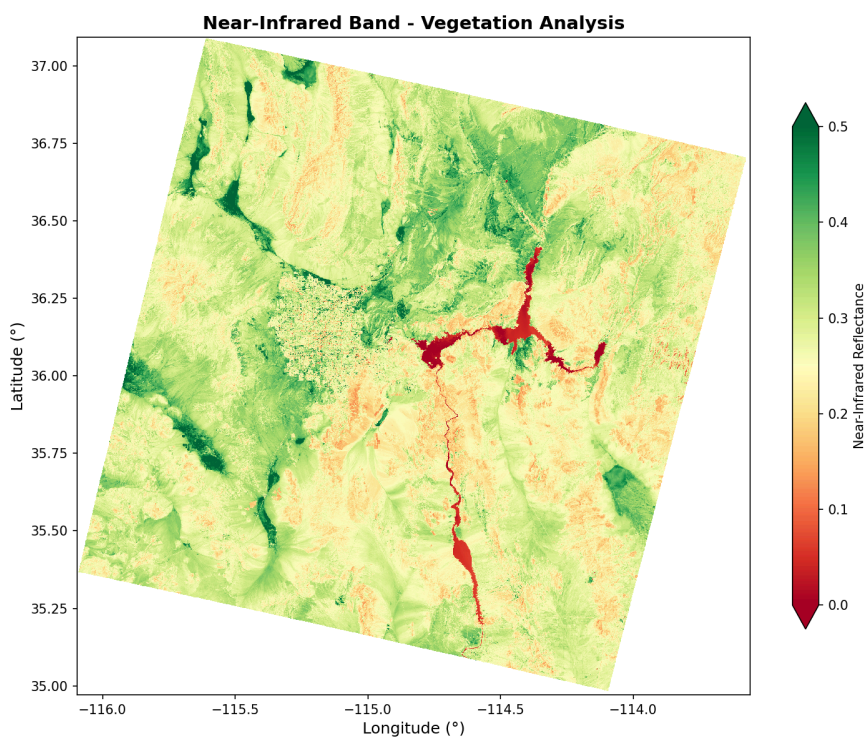


Figure 38: A near-infrared band of the Landsat 9 image of the Las Vegas area.

19.7.3. Overlaying Vector Data on Raster Images

Combining raster and vector data in visualizations provides important spatial context. For example, we can overlay a vector boundary on a Landsat image to see the city boundaries ([Figure 39](#)).

```
# Create a visualization with vector overlay
fig, ax = plt.subplots(figsize=(8, 6))

# Plot raster as background (single band for clarity)
data.attrs["long_name"] = "Surface Reflectance"
single_band = data.sel(band=4) # Red band
single_band.plot.imshow(
    ax=ax,
    vmin=0,
    vmax=0.4,
    cmap="gray",
    add_colorbar=True,
    cbar_kwargs={"label": "Red Band Reflectance"},
)

# Overlay vector boundary
if "boundary" in locals():
    boundary.to_crs(data.rio.crs).boundary.plot(ax=ax, color="red", linewidth=2)

plt.title("Landsat Image with Administrative Boundary", fontsize=14,
fontweight="bold")
plt.xlabel("Easting (m)", fontsize=12)
plt.ylabel("Northing (m)", fontsize=12)
plt.tight_layout()
plt.show()
```

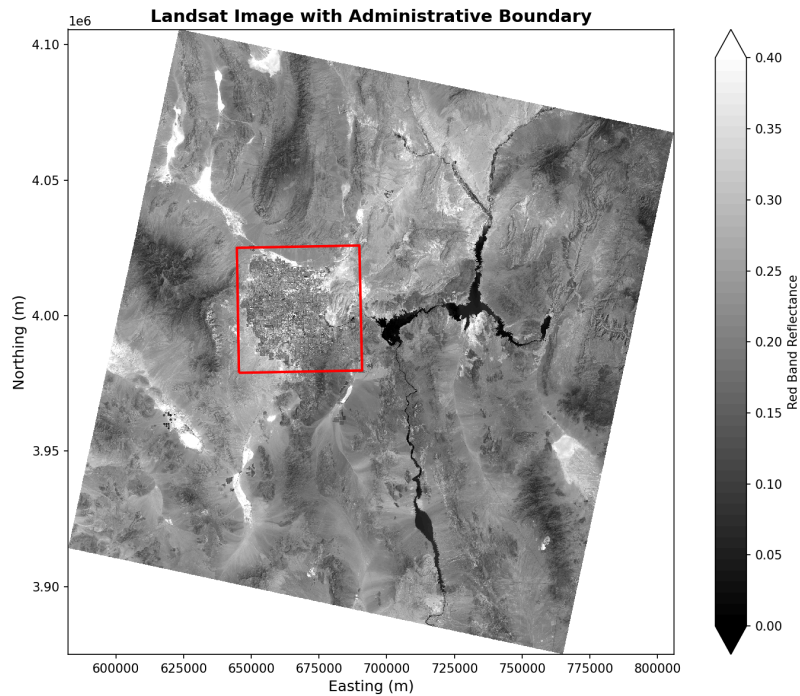


Figure 39: A Landsat 9 image of the Las Vegas area with a vector boundary overlay.

19.8. Data Storage and File Management

Proper data management is essential for reproducible geospatial analysis. Rioxarray provides robust tools for saving processed data while preserving all spatial metadata.

19.8.1. Saving Processed Raster Data

After processing, you'll often need to save results for future use or sharing. Rioxarray makes this easy with the `rio.to_raster()` method. For example, we can save the clipped Landsat image to a new file easily using the `rio.to_raster()` method:

```
# Save processed data as GeoTIFF (most common geospatial raster format)
output_filename = "las_vegas_landsat_processed.tif"

# Add metadata before saving
clipped_data.attrs["processing_date"] = str(np.datetime64("now"))
clipped_data.attrs["processed_by"] = "rioxarray tutorial"
clipped_data.attrs["description"] = "Landsat image clipped to Las Vegas area"

print(f"Saving processed data to {output_filename}...")
clipped_data.rio.to_raster(output_filename)
print("File saved successfully!")

# Verify the saved file
```

```

saved_data = rioarray.open_rasterio(output_filename)
print(f"Verified: Saved file has shape {saved_data.shape} and CRS {saved_data.rio.crs}")

```

If you want to save the data as a Cloud Optimized GeoTIFF (COG), you can use the `rio.to_raster()` method with the `driver="COG"` option. This will create a COG file that is optimized for cloud storage and can be accessed efficiently.

```

output_filename = "las_vegas_landsat_processed_cog.tif"
clipped_data.rio.to_raster(output_filename, driver="COG")

```

Best practices for saving raster data:

- **Use GeoTIFF format:** Widely supported and preserves all spatial metadata
- **Include metadata:** Document processing steps and data sources
- **Choose appropriate data types:** Balance precision with file size
- **Organize files:** Use descriptive names and consistent directory structures

19.8.2. Managing NoData Values

NoData values represent missing or invalid data in raster datasets, and proper handling of these values is crucial for accurate analysis. NoData values can occur for various reasons:

Common sources of NoData values:

- **Sensor limitations:** Areas outside the sensor's field of view or where the sensor malfunctioned
- **Atmospheric conditions:** Clouds, smoke, or atmospheric interference that prevents reliable measurements
- **Processing artifacts:** Areas masked out during data processing or quality control
- **Geographic constraints:** Areas where data collection is impossible (e.g., outside satellite orbits)
- **User-defined masks:** Areas intentionally excluded from analysis (e.g., water bodies in vegetation studies)

Rioarray provides comprehensive tools to handle NoData values effectively throughout your analysis workflow. First, you can check the current NoData value using the `rio.nodata` attribute. Then, you can set a new NoData value using the `rio.set_nodata()` method if needed.

```

# Check for existing NoData values
print("Current NoData value:", data.rio.nodata)

# Set NoData value if needed
data_with_nodata = data.rio.set_nodata(-9999)
print("Updated NoData value:", data_with_nodata.rio.nodata)

```

Sometimes, you need to mask out certain values in your raster data. For example, you might want to mask out very low pixels values that are likely to be invalid measurements. First, you can use the `where` method to mask out the values you don't want. Then, you can set the NoData value using the `rio.set_nodata()` method.


```
# Example: Mask out extreme values as NoData
# This is useful for removing outliers or invalid measurements
masked_data = data.where(data < 1.0, -9999) # Mask very high reflectance values
masked_data = masked_data.rio.set_nodata(-9999)
print(f"Applied masking: {np.sum(masked_data == -9999).values} pixels set to NoData")
```

19.9. Coordinate System Comparisons

Understanding how different coordinate systems affect your data is crucial for proper analysis. Each coordinate reference system (CRS) represents Earth's curved surface on a flat map using different mathematical projections, and each projection involves trade-offs between accuracy in area, distance, direction, and shape.

Key coordinate system concepts:

Geographic Coordinate Systems (like EPSG:4326 - WGS84):

- Use latitude and longitude in decimal degrees
- Excellent for global datasets and navigation
- Coordinates are intuitive (e.g., 40.7589°N, 73.9851°W for New York City)
- However, not suitable for accurate area or distance calculations
- Grid cells vary in actual size depending on latitude

Projected Coordinate Systems (like UTM zones):

- Use linear units (meters or feet) in X,Y coordinates
- Designed to minimize distortion within specific regions
- Ideal for accurate measurements and area calculations
- Coordinates are less intuitive but more suitable for analysis
- Grid cells have consistent size within the projection area

Web Mercator (EPSG:3857):

- Optimized for web mapping applications
- Preserves angles and shapes reasonably well
- Severely distorts areas, especially near the poles
- Used by most online mapping services (Google Maps, OpenStreetMap)

```
# Compare the same data in different coordinate systems
print("Comparing coordinate systems...")

# Web Mercator (common for web mapping)
mercator_data = data.rio.reproject("EPSG:3857")
print(f"Web Mercator (EPSG:3857): {mercator_data.rio.crs}")
print(
    f"    Coordinate range: X: {mercator_data.x.min().values:.0f} to {mercator_data.x.max().values:.0f}"
)
print(
```



```

        f"                                Y: {mercator_data.y.min().values:.0f} to
{mercator_data.y.max().values:.0f}"
    )

    # Geographic coordinates (WGS84)
    print(f"\nGeographic (EPSG:4326): {data_geographic.rio.crs}")
    print(
        f"    Coordinate range: Lon: {data_geographic.x.min().values:.3f} to
{data_geographic.x.max().values:.3f}"
    )
    print(
        f"                                Lat: {data_geographic.y.min().values:.3f} to
{data_geographic.y.max().values:.3f}"
    )

```

```

Comparing coordinate systems...
Web Mercator (EPSG:3857): EPSG:3857
    Coordinate range: X: -12923800 to -12641328
                      Y: 4160003 to 4452058

Geographic (EPSG:4326): EPSG:4326
    Coordinate range: Lon: -116.097 to -113.559
                      Lat: 34.972 to 37.093

```

19.9.1. Side-by-Side Coordinate System Visualization

The example below compares how the same satellite image appears when displayed in two different coordinate reference systems (CRS): geographic coordinates (EPSG:4326) and Web Mercator (EPSG:3857) (see [Figure 40](#)).

```

# Create side-by-side comparison of different projections
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

# Plot in geographic coordinates
data_geographic.sel(band=[4, 3, 2]).plot.imshow(ax=ax1, vmin=0, vmax=0.3)
ax1.set_title("Geographic Coordinates (EPSG:4326)")
ax1.set_xlabel("Longitude (°)")
ax1.set_ylabel("Latitude (°)")

# Plot in Web Mercator
mercator_data.sel(band=[4, 3, 2]).plot.imshow(ax=ax2, vmin=0, vmax=0.3)
ax2.set_title("Web Mercator (EPSG:3857)")
ax2.set_xlabel("Easting (m)")
ax2.set_ylabel("Northing (m)")

```

```
plt.tight_layout()
plt.show()
```

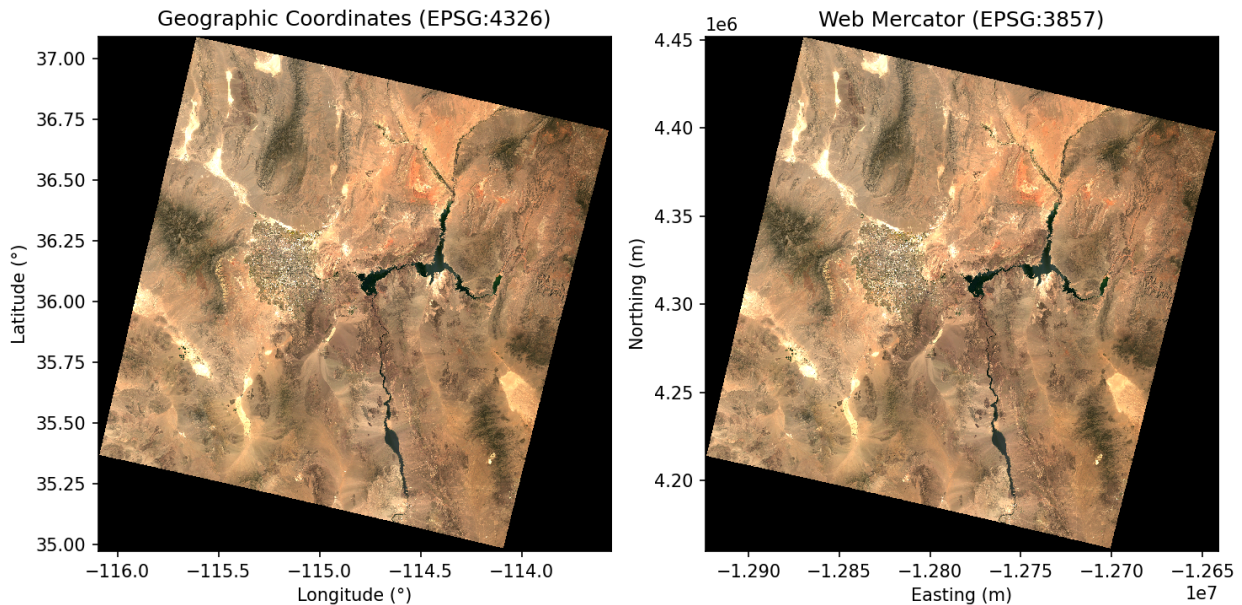


Figure 40: A comparison of the same data in different coordinate systems.

19.10. Introduction to Band Math

Band math involves combining different spectral bands to derive new information about Earth's surface features. This is one of the most powerful techniques in remote sensing analysis, allowing us to extract insights that aren't visible in individual bands alone.

19.10.1. Computing Vegetation Indices

The Normalized Difference Vegetation Index (NDVI) is one of the most widely used vegetation indices in remote sensing. Understanding how and why NDVI works is crucial for anyone working with satellite imagery.

The science behind NDVI:

NDVI takes advantage of the unique spectral properties of vegetation:

- **Healthy vegetation** strongly absorbs red light (for photosynthesis) while strongly reflecting near-infrared light (which can damage plant tissues if absorbed)
- **Stressed or sparse vegetation** reflects more red light and less near-infrared light
- **Non-vegetated surfaces** (soil, water, urban areas) have different spectral signatures that result in lower NDVI values

NDVI formula and interpretation:

$$\text{NDVI} = (\text{NIR} - \text{Red}) / (\text{NIR} + \text{Red})$$

The normalization (dividing by the sum) ensures that NDVI values always fall between -1 and $+1$, making it easy to compare across different images and time periods. This normalization also helps reduce the effects of atmospheric conditions and sun angle variations.

Why NDVI is so useful:

- **Simple and robust:** Easy to calculate and interpret across different sensors and time periods
- **Standardized scale:** Values between -1 and $+1$ make comparison straightforward
- **Physically meaningful:** Directly related to photosynthetic activity and biomass
- **Widely validated:** Decades of research have established its reliability for vegetation monitoring

The example below shows how to compute NDVI from Landsat 9 data using `rioxarray`:

```
# Calculate NDVI using the standard formula: (NIR - Red) / (NIR + Red)
print("Calculating NDVI...")

# Extract the required bands
red_band = data_geographic.sel(band=4) # Red band
nir_band = data_geographic.sel(band=5) # Near-infrared band

# Calculate NDVI with proper handling of division by zero
ndvi = (nir_band - red_band) / (nir_band + red_band)

# Clip values to valid NDVI range and handle any invalid calculations
ndvi = ndvi.clip(min=-1, max=1)
ndvi = ndvi.where(np.isfinite(ndvi)) # Remove infinite values

# Add descriptive metadata
ndvi.attrs["long_name"] = "Normalized Difference Vegetation Index"
ndvi.attrs["valid_range"] = "[-1, 1]"
ndvi.attrs["description"] = "NDVI = (NIR - Red) / (NIR + Red)"

print(f"NDVI statistics:")
print(f"  Range: {ndvi.min().values:.3f} to {ndvi.max().values:.3f}")
print(f"  Mean: {ndvi.mean().values:.3f}")
```

19.10.2. Visualizing Vegetation Analysis

As demonstrated in the previous section, NDVI is a powerful tool for analyzing vegetation patterns in satellite imagery. The example below shows how to visualize NDVI and extract vegetation information from the Landsat 9 data (see [Figure 41](#)).

```
# Create a comprehensive NDVI visualization
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

# Plot raw NDVI
ndvi.plot.imshow(
    ax=ax1,
    cmap="RdYlGn",
```

```

    vmin=-0.5,
    vmax=0.8,
    add_colorbar=True,
    cbar_kwargs={"label": "NDVI Value"},
)
ax1.set_title("Raw NDVI Values", fontsize=14, fontweight="bold")
ax1.set_xlabel("Longitude (°)")
ax1.set_ylabel("Latitude (°)")

# Plot vegetation areas only (NDVI > 0.2)
vegetation_mask = ndvi.where(ndvi > 0.2)
vegetation_mask.plot.imshow(
    ax=ax2,
    cmap="Greens",
    vmin=0.2,
    vmax=0.6,
    add_colorbar=True,
    cbar_kwargs={"label": "NDVI Value (Vegetation Only)"},
)
ax2.set_title("Vegetation Areas (NDVI > 0.2)", fontsize=14, fontweight="bold")
ax2.set_xlabel("Longitude (°)")
ax2.set_ylabel("Latitude (°)")

plt.tight_layout()
plt.show()

```

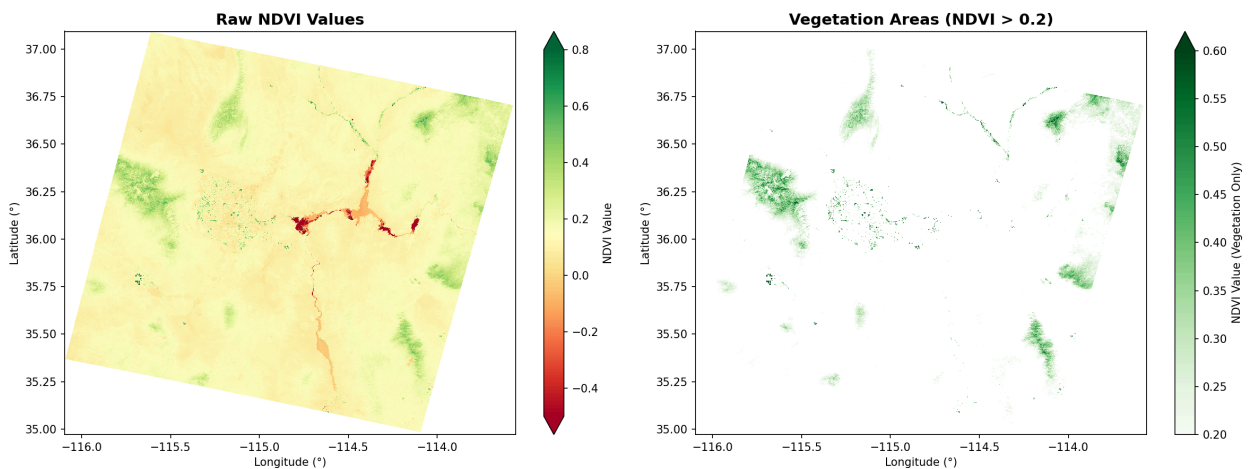


Figure 41: A visualization of the NDVI of the Landsat 9 image of the Las Vegas area.

19.10.3. Interpreting NDVI Results

This example below analyzes the distribution of land cover types using NDVI values by categorizing pixels into vegetation (NDVI > 0.2), water (NDVI < 0), and urban or bare soil (NDVI between 0 and 0.2). It calculates the total number of pixels in the NDVI dataset and counts how many fall into each

category, then prints the results along with their percentage of the total. This provides a quick overview of vegetation presence and other surface types in the analyzed area.

```
# Analyze vegetation distribution
total_pixels = ndvi.size
vegetation_pixels = np.sum(ndvi > 0.2).values
water_pixels = np.sum(ndvi < 0).values
urban_pixels = np.sum((ndvi >= 0) & (ndvi <= 0.2)).values

print("NDVI Analysis Results:")
print(f"  Total pixels analyzed: {total_pixels:,}")
print(
    f"  Vegetation pixels (NDVI > 0.2): {vegetation_pixels:,} "
    f"({100*vegetation_pixels/total_pixels:.1f}%) "
)
print(
    f"  Water pixels (NDVI < 0): {water_pixels:,} ({100*water_pixels/"
    f"total_pixels:.1f}%) "
)
print(
    f"  Urban/Bare soil (0 ≤ NDVI ≤ 0.2): {urban_pixels:,} ({100*urban_pixels/"
    f"total_pixels:.1f}%) "
)
```

```
NDVI Analysis Results:
  Total pixels analyzed: 6,611,012
  Vegetation pixels (NDVI > 0.2): 443,905 (6.7%)
  Water pixels (NDVI < 0): 49,862 (0.8%)
  Urban/Bare soil (0 ≤ NDVI ≤ 0.2): 4,000,320 (60.5%)
```

NDVI interpretation:

- **NDVI > 0.4:** Dense vegetation (forests, healthy crops)
- **0.2 < NDVI < 0.4:** Moderate vegetation (grasslands, sparse vegetation)
- **0 < NDVI < 0.2:** Bare soil, urban areas, sparse vegetation
- **NDVI < 0:** Water, snow, clouds

19.11. Key Takeaways

This chapter introduced the core concepts and capabilities of `rioxarray` for geospatial raster analysis, highlighting its seamless integration of `Xarray`'s array handling with `Rasterio`'s geospatial functionality. `Rioxarray` maintains spatial awareness through coordinate reference systems and georeferencing, supports efficient operations like reprojection and clipping, and preserves metadata throughout processing. It offers flexible visualization with `matplotlib` and broad format compatibility, making it a powerful tool for producing reproducible and spatially aware analyses. By simplifying complex workflows while retaining the richness of geospatial data, `rioxarray` lays a solid foundation for advanced applications such as time series analysis, machine learning, and remote sensing.

19.12. Exercises

The following exercises will help you practice essential rioarray concepts using a different dataset and various analysis scenarios.

19.12.1. Sample Dataset

For the exercises, we will use a sample GeoTIFF raster dataset of Libya, which is available at the following URL: <https://github.com/opengeos/datasets/releases/download/raster/Libya-2023-09-13.tif>

19.12.2. Exercise 1: Load and Inspect a Raster Dataset

Practice loading and exploring the fundamental properties of a georeferenced raster dataset.

1. Use `rioarray` to load the Libya GeoTIFF raster file from the provided URL.
2. Inspect the dataset by examining and printing its dimensions, coordinates, and key attributes.
3. Check and print the coordinate reference system (CRS) and affine transformation of the dataset.
4. Calculate and display the spatial resolution and total geographic extent of the dataset.

19.12.3. Exercise 2: Coordinate System Transformation

Learn to reproject raster data between different coordinate reference systems.

1. Reproject the loaded Libya raster dataset from its original CRS to EPSG:4326 (WGS84 Geographic).
2. Compare the coordinate ranges and spatial extents before and after reprojection.
3. Create side-by-side visualizations of the original and reprojected datasets to observe the differences.
4. Calculate how the pixel size changed after reprojection.

19.12.4. Exercise 3: Spatial Subsetting with Bounding Boxes

Practice extracting spatial subsets using coordinate-based clipping.

1. Define a bounding box that covers a specific region of Libya (you can choose any area that interests you).
2. Clip the raster dataset using this bounding box.
3. Compare the sizes of the original and clipped datasets.
4. Create a visualization showing the clipped area with appropriate titles and labels.

19.12.5. Exercise 4: Vector-Based Masking

Learn to use vector data to mask raster datasets for precise spatial analysis.

1. Load the GeoJSON file at https://github.com/opengeos/datasets/releases/download/raster/Derna_Libya.geojson using `geopandas`.

2. Examine the vector data and ensure it has a proper coordinate reference system.
3. Use the GeoJSON polygon to mask the Libya raster dataset, keeping only the data within the polygon boundaries.
4. Create a visualization showing the masked raster data with the vector boundary overlay.

19.12.6. Exercise 5: Resolution Analysis and Data Export

Practice resampling raster data and saving processed results.

1. Resample the Libya raster dataset to a 300-meter resolution using an appropriate resampling method.
2. Compare the dimensions and file sizes before and after resampling.
3. Add meaningful metadata attributes to document your processing steps.
4. Save the resampled raster dataset as a new GeoTIFF file with a descriptive filename.
5. Verify your saved file by loading it back and confirming it preserved the spatial reference information.

Chapter 20. Interactive Visualization with Leafmap

20.1. Introduction

In the modern era of geospatial analysis, interactive web-based maps have become essential tools for exploring, analyzing, and communicating spatial information. Traditional static maps, while useful for certain purposes, lack the dynamic capabilities needed for interactive data exploration, real-time analysis, and engaging presentations. This is where [Leafmap](https://leafmap.org)⁵⁹ comes in—a powerful, open-source Python package that bridges the gap between complex geospatial data and intuitive, interactive visualization.

What is Leafmap?

Leafmap is fundamentally designed to make geospatial visualization accessible to everyone, from beginners learning GIS programming to experienced geospatial professionals working with complex datasets. Think of it as a Swiss Army knife for interactive mapping in Python—it provides a unified, simplified interface that works with multiple mapping libraries behind the scenes, allowing you to focus on your data and analysis rather than wrestling with complex API syntax.

The Challenge Leafmap Solves

Before Leafmap, creating interactive maps in Python often required deep knowledge of various mapping libraries, each with their own syntax, capabilities, and limitations. You might need to use [folium](https://python-visualization.github.io/folium)⁶⁰ for web-based maps, [ipyleaflet](https://ipyleaflet.readthedocs.io)⁶¹ for Jupyter integration, [bokeh](https://docs.bokeh.org/en/latest/docs/user_guide/topics/geo.html)⁶² for dashboard applications, [plotly](https://plotly.com/python/maps)⁶³ for statistical visualizations, and [maplibre](https://eoda-dev.github.io/py-maplibregl)⁶⁴ for 3D visualization. Each library serves its purpose, but learning and switching between them creates friction in your workflow.

Leafmap elegantly solves this problem by building on top of these established libraries—including maplibre, folium, ipyleaflet, bokeh, plotly, and pydeck—and extending their functionalities with a unified, intuitive API. This means you can leverage the strengths of multiple mapping ecosystems through a single, consistent interface.

Why Interactive Maps Matter in Geospatial Analysis

Interactive maps are powerful analytical tools that enable exploratory data analysis, multi-scale visualization, and real-time feedback. Unlike static maps, they allow you to zoom, pan, and interact with your data to discover patterns that might otherwise remain hidden. You can seamlessly navigate from global overviews to detailed local insights, turn different data layers on and off to understand relationships, and immediately see the results of data processing operations. This interactivity also enhances communication by allowing stakeholders to explore the data themselves through engaging presentations.

This chapter will guide you through Leafmap’s essential features, focusing on the fundamental concepts and techniques you need to create effective interactive geospatial visualizations. We’ll start with the basics of map creation and customization, then progress through data visualization techniques for both vector and raster data. Each section builds upon previous concepts while providing practical examples that you can immediately apply to your own geospatial projects.

⁵⁹<https://leafmap.org>

⁶⁰<https://python-visualization.github.io/folium>

⁶¹<https://ipyleaflet.readthedocs.io>

⁶²https://docs.bokeh.org/en/latest/docs/user_guide/topics/geo.html

⁶³<https://plotly.com/python/maps>

⁶⁴<https://eoda-dev.github.io/py-maplibregl>

20.2. Learning Objectives

By the end of this chapter, you will be able to:

- **Create and customize interactive maps** using Leafmap’s intuitive interface, including setting map properties like center location, zoom level, and map controls
- **Add and configure different basemap layers** to provide appropriate context for your geospatial data, including standard web map services and custom tile layers
- **Visualize and style vector geospatial data** effectively, including points, lines, and polygons from various formats like GeoJSON, Shapefile, and GeoParquet
- **Display and analyze raster data** including satellite imagery, digital elevation models, and other gridded datasets using modern cloud-optimized formats
- **Apply fundamental interactive mapping techniques** such as layer management, legends, and basic map controls that enhance user experience and data interpretation

20.3. Installing and Setting Up Leafmap

Before diving into interactive mapping, let’s set up Leafmap in your Python environment. The installation process is straightforward, and we’ll cover the essential setup steps to get you started.

20.3.1. Installation Methods

Leafmap can be installed using either conda or pip, depending on your preferred package management system. For geospatial work, we generally recommend conda because it handles complex geospatial library dependencies more reliably.

Installation via conda (Recommended)

```
# %conda install -c conda-forge leafmap
```

Installation via pip

```
# %pip install -U leafmap
```

Once installed, you can import Leafmap into your Python environment:

```
import leafmap
```

20.3.2. Understanding Leafmap’s Backend Architecture

One of Leafmap’s most powerful features is its support for multiple plotting backends. Think of backends as different “engines” that power your interactive maps, each with their own strengths and use cases. The default **ipyleaflet** backend provides the most comprehensive feature set and best integration with Jupyter environments. **Folium** excels for web-based applications and standalone HTML map exports, while **maplibre** offers modern, performant vector tile rendering.

For this chapter, we’ll primarily use the default **ipyleaflet** backend, which offers the best balance of features and ease of use for learning fundamental concepts. We will cover the maplibre backend in another chapter.

To switch backends when needed, you can import the specific backend module:

```
# import leafmap.foliummap as leafmap # For folium backend
# import leafmap.maplibregl as leafmap # For MapLibre backend
```

20.4. Creating Interactive Maps

Now that Leafmap is set up, let's create your first interactive map. Understanding the basics of map creation and customization forms the foundation for all subsequent geospatial visualization work.

20.4.1. Your First Interactive Map

Creating a basic interactive map with Leafmap is remarkably simple. The `Map()` function creates a map object that you can display and interact with directly in your Jupyter environment:

```
m = leafmap.Map()
m
```

When you run this code, you'll see an interactive map centered on a default location with OpenStreetMap as the base layer. You can immediately pan by clicking and dragging, zoom using the mouse wheel or zoom controls, switch to fullscreen using the fullscreen button, and access the toolbar for additional functionality. Note that the fullscreen button will not work in VS Code. Try JupyterLab instead.

Understanding the Map Object

The variable `m` now contains a map object that represents your interactive map. This object is your interface for adding data, changing settings, and customizing the map's appearance and behavior. Think of it as a canvas that you'll progressively build upon throughout your analysis.

20.4.2. Customizing Map Properties

While the default map is functional, you'll often want to customize it for your specific needs. The most common customizations involve setting the initial view and map dimensions.

20.4.2.1. Setting Center Location and Zoom Level

The `center` parameter takes a tuple of (latitude, longitude) coordinates, while `zoom` controls the initial scale. Understanding zoom levels is crucial: low zoom levels (1-4) provide continental or global views, medium zoom levels (5-10) show regional or state-level detail, and high zoom levels (11-18) display city, neighborhood, or building-level detail. The default map height is 600 pixels. Adjust the height as needed.

```
# Center the map on the continental United States
m = leafmap.Map(center=(40, -100), zoom=4, height="600px")
m
```

20.4.3. Managing Map Controls

Interactive maps include various controls that help users navigate and interact with the data. Understanding these controls and when to show or hide them is essential for creating effective user experiences.

20.4.3.1. Understanding Default Controls

By default, Leafmap maps include zoom controls (+ and - buttons), a fullscreen control, a scale control showing the map scale at the current zoom level, attribution credits for map data and tile providers, and a toolbar control for accessing additional Leafmap functionality.

20.4.3.2. Customizing Control Visibility

You can selectively enable or disable controls based on your needs:

```
# Create a minimal map with most controls disabled
m = leafmap.Map(
    center=(40, -100),
    zoom=4,
    zoom_control=False, # Remove zoom buttons
    draw_control=False, # Remove drawing tools
    scale_control=False, # Remove scale indicator
    fullscreen_control=False, # Remove fullscreen button
    attribution_control=False, # Remove attribution text
    toolbar_control=False, # Remove Leafmap toolbar
)
m
```

20.4.3.3. Adding Search Functionality

Search functionality allows users to quickly navigate to specific locations, making maps more user-friendly for exploratory analysis. The example below adds a search control to the map (see [Figure 42](#)):

```
# Create a new map for demonstration
m = leafmap.Map(center=(40, -100), zoom=4, draw_control=False, height="500px")

# Add search control using OpenStreetMap's Nominatim service
url = "https://nominatim.openstreetmap.org/search?format=json&q={s}"
m.add_search_control(url, zoom=10, position="topleft")
m
```

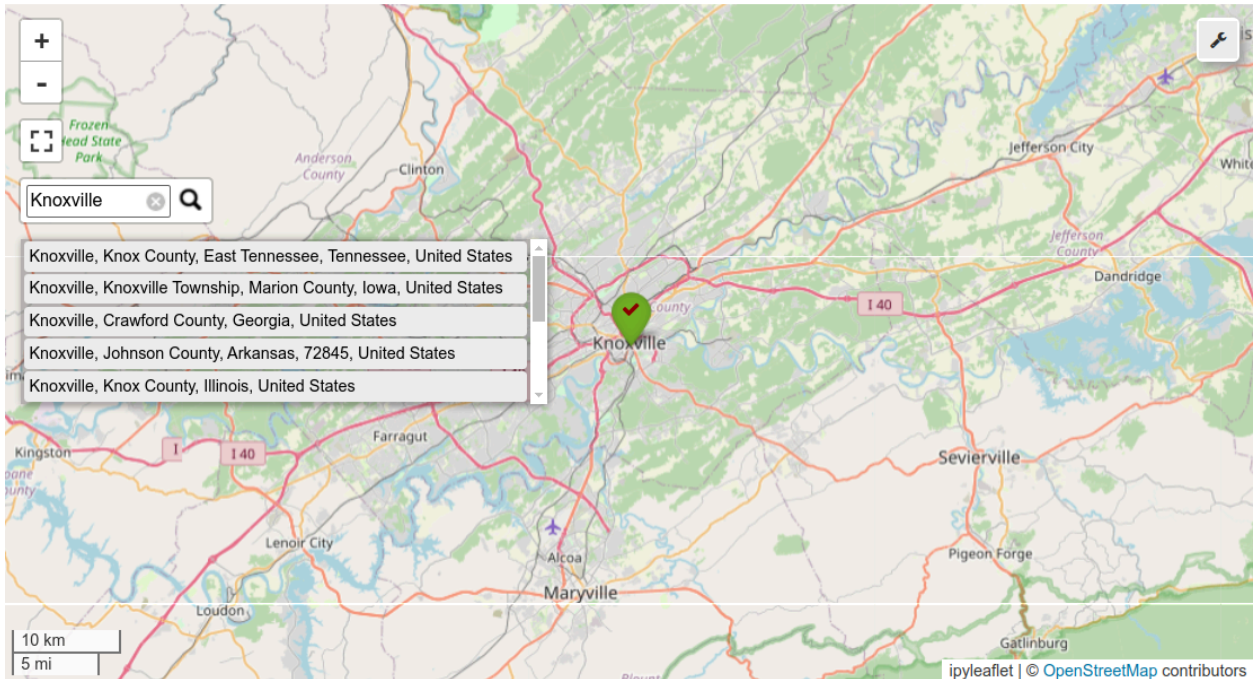


Figure 42: The leafmap search control for searching places on the map.

20.4.4. Working with Map Layers

Understanding how to manage map layers is fundamental to effective interactive mapping. Layers are the building blocks of your map—each dataset, basemap, or visualization element exists as a separate layer that can be controlled independently.

20.4.4.1. Accessing and Inspecting Layers

Every map object maintains a collection of layers that you can inspect and manipulate:

```
# Create a simple map and examine its layers
m = leafmap.Map()
print(f"Number of layers: {len(m.layers)}")
print(f"Layer types: {[type(layer).__name__ for layer in m.layers]}")
```

20.4.4.2. Layer Management Operations

You can add and remove layers programmatically, which is useful for dynamic map updates:

```
# Remove the last layer (be careful with this operation)
if len(m.layers) > 1: # Keep at least one layer
    m.remove(m.layers[-1])

# Check the remaining layers
print(f"Remaining layers: {len(m.layers)}")
```

20.4.4.3. Clearing Map Content

Sometimes you need to start fresh or create minimal maps for specific purposes:

```
# Create a map with default content
m = leafmap.Map()

# Remove all interactive controls
m.clear_controls()

# Remove all layers (this will result in a blank map)
m.clear_layers()

# Display the minimal map
m
```

20.5. Changing Basemaps

Basemaps provide the visual foundation for your geospatial data, offering context and reference information that helps users understand and interpret your analysis. Think of basemaps as the backdrop against which your data tells its story—choosing the right basemap can dramatically improve the clarity and effectiveness of your visualization.

20.5.1. Understanding Basemaps and Their Role

A basemap is essentially a background layer that provides geographic context for your data layers. Satellite imagery shows actual Earth surface features and works well for environmental analysis, street maps emphasize roads and urban features for transportation and urban planning, topographic maps highlight elevation and terrain for outdoor applications, and minimalist maps reduce visual clutter to emphasize your data layers.

20.5.2. Adding Predefined Basemaps

Leafmap provides access to numerous high-quality basemap services. Here's how to add a topographic basemap that shows elevation contours and terrain features ([Figure 43](#)):

```
m = leafmap.Map()
m.add_basemap("OpenTopoMap")
m
```

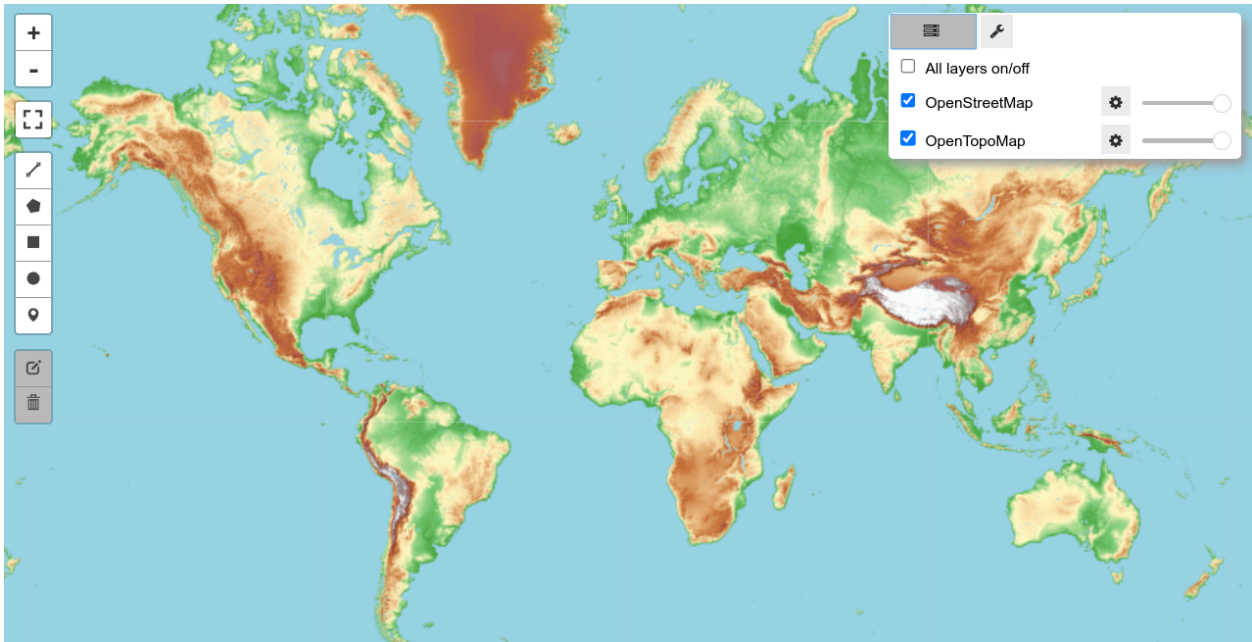


Figure 43: The OpenTopoMap basemap.

Popular Basemap Options

Common basemaps include "OpenStreetMap" for standard street maps with good detail, "OpenTopoMap" for topographic features and elevation contours, "Esri.WorldImagery" for high-resolution satellite imagery, and "CartoDB.Positron" for clean minimalist design.

20.5.3. Interactive Basemap Selection

For exploratory analysis, you might want to quickly switch between different basemaps to see which provides the best context for your data:

```
m = leafmap.Map()
m.add_basemap_gui()
m
```

20.5.4. Adding Custom XYZ Tile Layers

XYZ tile layers are one of the most common formats for web-based mapping services. The XYZ format organizes map tiles in a pyramid structure where each tile is identified by three coordinates: X (column), Y (row), and Z (zoom level). This system allows for efficient loading and display of map data across different zoom levels.

Understanding XYZ Tile Structure

The XYZ tiling scheme divides the world into a grid at each zoom level. At zoom 0, the entire world fits in a single 256256 pixel tile. At zoom 1, the world is divided into 4 tiles (22 grid), at zoom 2 into 16 tiles (4*4 grid), and so on, with each zoom level quadrupling the number of tiles.

Here's how to add a custom XYZ tile layer, using Google Satellite imagery as an example:

```
m = leafmap.Map()
m.add_tile_layer(
    url="https://mt1.google.com/vt/lyrs=y&x={x}&y={y}&z={z}",
    name="Google Satellite",
    attribution="Google",
)
m
```

Key Parameters for XYZ Layers

The `url` parameter contains the URL template with {x}, {y}, and {z} placeholders, `name` sets the display name for layer controls, and `attribution` provides credit text for the data provider (important for legal compliance).

20.5.5. Adding Web Map Service (WMS) Layers

Web Map Service (WMS) is an Open Geospatial Consortium (OGC) standard that provides a standardized way to request map images from geospatial databases. Unlike XYZ tiles which are pre-rendered, WMS layers are generated on-demand, allowing for dynamic styling and querying capabilities. Here's how to add a WMS layer showing high-resolution aerial imagery:

```
m = leafmap.Map(center=[40, -100], zoom=4)
url = "https://imagery.nationalmap.gov/arcgis/services/USGSNAIPPlus/ImageServer/WMSServer?"
m.add_wms_layer(
    url=url,
    layers="USGSNAIPPlus",
    name="NAIP",
    attribution="USGS",
    format="image/png",
    shown=True,
)
m
```

Understanding WMS Parameters

- `url` : The base URL of the WMS service (must end with '?' or '&')
- `layers` : The specific layer name(s) from the WMS service
- `name` : Display name for your map layer
- `attribution` : Data source attribution
- `format` : Image format for the returned tiles (PNG for transparency, JPEG for smaller file sizes)
- `shown` : Whether the layer is visible by default

One great source of WMS layers is the [USGS National Map](https://apps.nationalmap.gov/services)⁶⁵. The USGS NAIP Plus service we used above is from the USGS National Map.

⁶⁵<https://apps.nationalmap.gov/services>

20.5.6. Adding Legends for Data Context

Legends are essential components that help users understand what your map symbols and colors represent. Without proper legends, even the most sophisticated geospatial visualization can be confusing and ineffective. Leafmap provides both built-in legends for common datasets and the ability to create custom legends.

Here's an example of adding a legend for the National Land Cover Database (NLCD), which classifies land use across the United States (see [Figure 44](#)):

```
m = leafmap.Map(center=[40, -100], zoom=4)
m.add_basemap("Esri.WorldImagery")
url = "https://www.mrlc.gov/geoserver/mrlc_display/NLCD_2021_Land_Cover_L48/wms?"
m.add_wms_layer(
    url=url,
    layers="NLCD_2021_Land_Cover_L48",
    name="NLCD 2021",
    attribution="MRLC",
    format="image/png",
    shown=True,
)
m.add_legend(title="NLCD Land Cover Type", builtin_legend="NLCD")
m
```

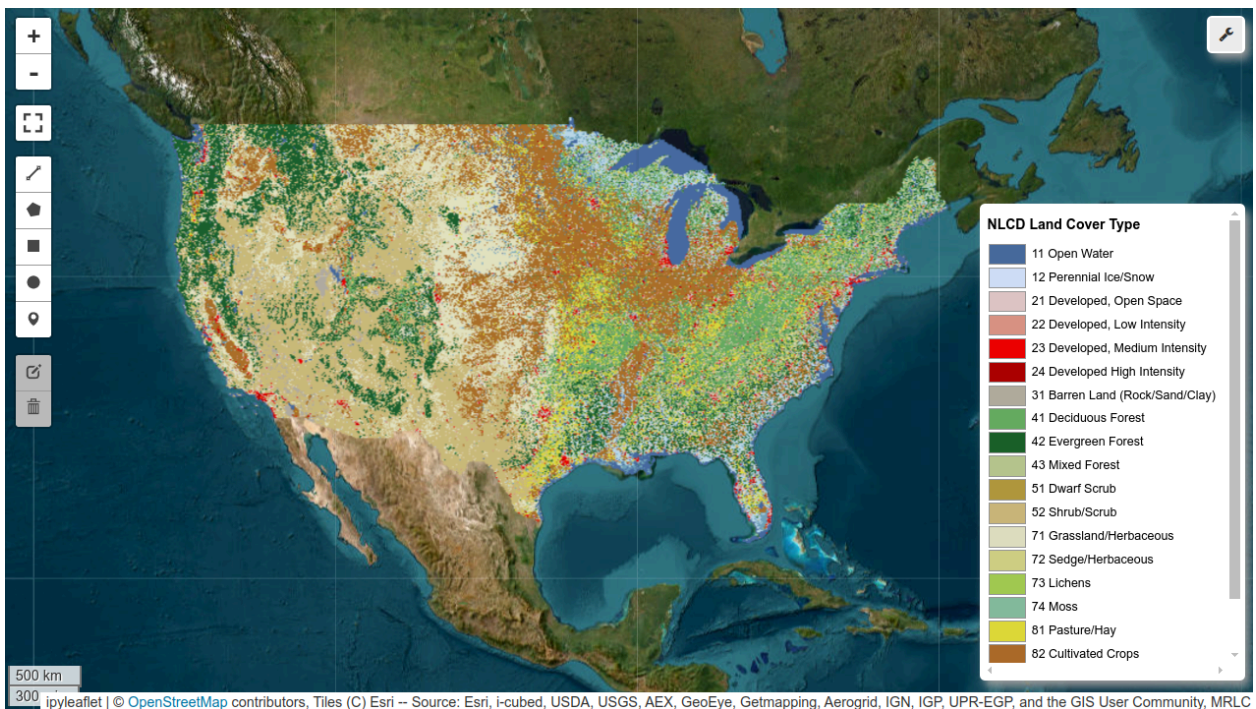


Figure 44: The NLCD legend.

20.5.7. Adding Colorbars for Continuous Data

While legends work well for categorical data (like land cover types), continuous data such as elevation, temperature, or precipitation requires different visualization approaches. Colorbars provide a continuous scale that shows how colors map to numerical values.

Here's how to add a colorbar for topographic data (see [Figure 45](#)):

```
m = leafmap.Map()
m.add_basemap("OpenTopoMap")
m.add_colormap(
    cmap="terrain",
    label="Elevation (meters)",
    orientation="horizontal",
    vmin=0,
    vmax=4000,
)
m
```

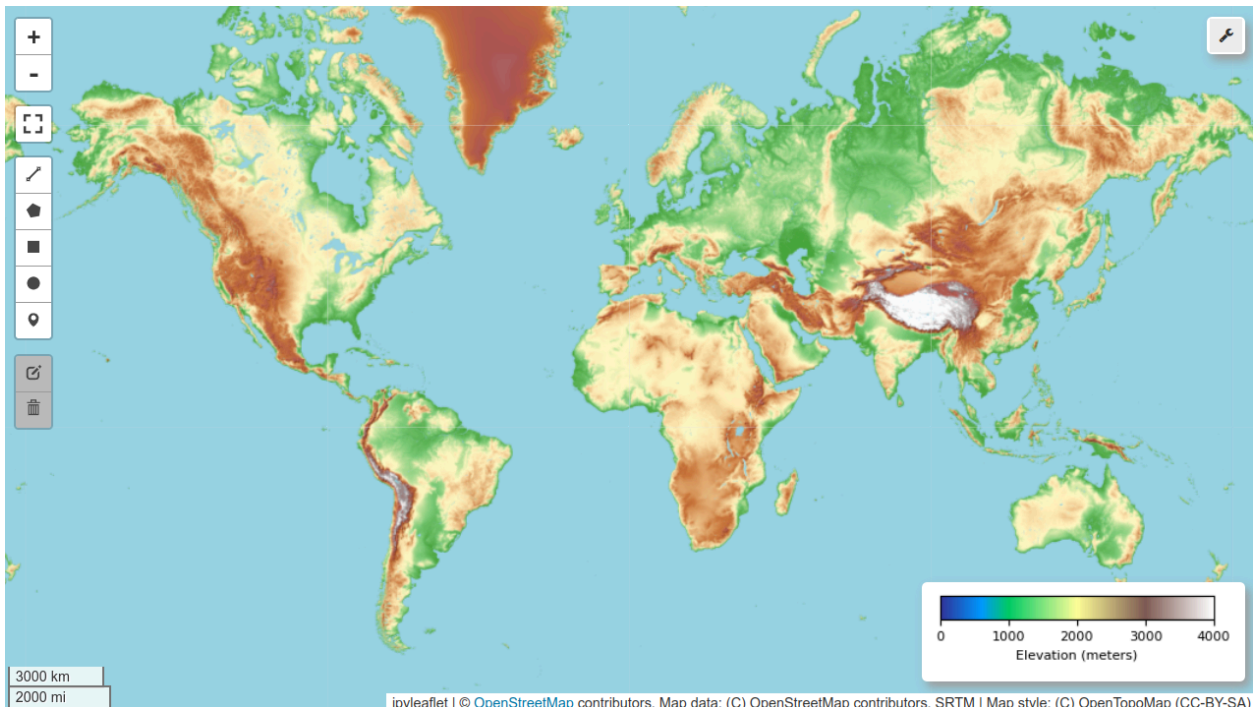


Figure 45: Adding a colorbar to a map.

You can use any `cmap` names from the [matplotlib colormap gallery](#)⁶⁶. The example above uses the "terrain" colormap. Here are some other popular colormaps for different data types:

- **Elevation:** "terrain", "viridis", "plasma"
- **Temperature:** "coolwarm", "RdYlBu"
- **Precipitation:** "Blues", "BuPu"

⁶⁶<https://matplotlib.org/stable/users/explain/colors/colormaps.html>

- **Population/Density:** "YlOrRd", "Reds"

To show the list of all colormaps, you can use the following code:

```
import leafmap.colormaps as cm

cm.plot_colormaps(width=8, height=0.3)
```

20.6. Visualizing Vector Data

Vector data represents discrete geographic features using points, lines, and polygons—the fundamental building blocks of geographic information systems. Unlike raster data which represents the world as a grid of cells, vector data captures the precise boundaries and locations of real-world features. This makes vector data ideal for representing cities, roads, property boundaries, administrative regions, and other discrete geographic entities.

Leafmap provides comprehensive support for visualizing vector data from various formats including GeoJSON, Shapefile, GeoPackage, and any format supported by [GeoPandas](#). The library offers intuitive methods for adding, styling, and interacting with vector features.

20.6.1. Working with Point Data: Markers

Point data is perhaps the most straightforward type of vector data to understand and visualize. Points represent specific locations on Earth's surface and are commonly used for landmarks, monitoring stations, events, or any phenomena that can be represented by a single coordinate pair.

20.6.1.1. Adding Individual Markers

The simplest way to add point data is through individual markers. Markers are interactive elements that can display information when clicked and can be made draggable for interactive applications:

```
m = leafmap.Map()
location = [40, -100] # [latitude, longitude]
m.add_marker(location, draggable=True)
m
```

20.6.1.2. Adding Multiple Markers Efficiently

When you have several point locations to display, adding them all at once is more efficient than adding them individually:

```
m = leafmap.Map()
# Coordinates for three different cities
locations = [
    [40, -100], # Central US
    [45, -110], # Northwestern US
    [50, -120], # Southwestern Canada
```

```
]
m.add_markers(markers=locations)
m
```

20.6.1.3. Managing Large Point Datasets with Clustering

When working with hundreds or thousands of points, individual markers can create visual clutter and performance issues. Marker clustering solves this by grouping nearby points into clusters that expand when users zoom in for detail (see [Figure 46](#)):

```
m = leafmap.Map()
url = "https://github.com/opengeos/datasets/releases/download/world/world_cities.csv"
m.add_marker_cluster(url, x="longitude", y="latitude", layer_name="World cities")
m
```

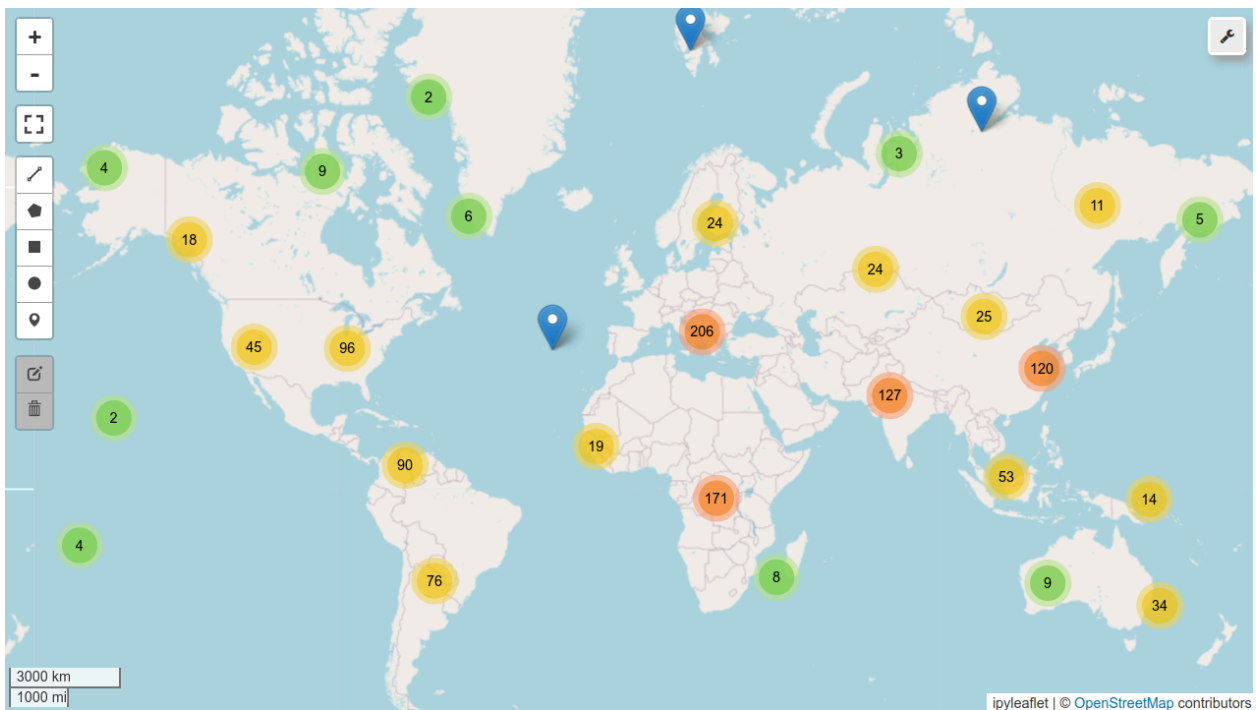


Figure 46: A marker cluster.

20.6.1.4. Advanced Marker Customization

Markers can be extensively customized with colors, icons, and labels to convey additional information. This example demonstrates styling points from a CSV file of U.S. cities, using different icons for different regions (see [Figure 47](#)):

```

m = leafmap.Map(center=[40, -100], zoom=4)
cities = "https://github.com/opengeos/datasets/releases/download/us/cities.csv"
regions = "https://github.com/opengeos/datasets/releases/download/us/us_regions.
geojson"
m.add_geojson(regions, layer_name="US Regions")
m.add_points_from_xy(
    cities,
    x="longitude",
    y="latitude",
    color_column="region",
    icon_names=["gear", "map", "leaf", "globe"],
    spin=True,
    add_legend=True,
)
m

```

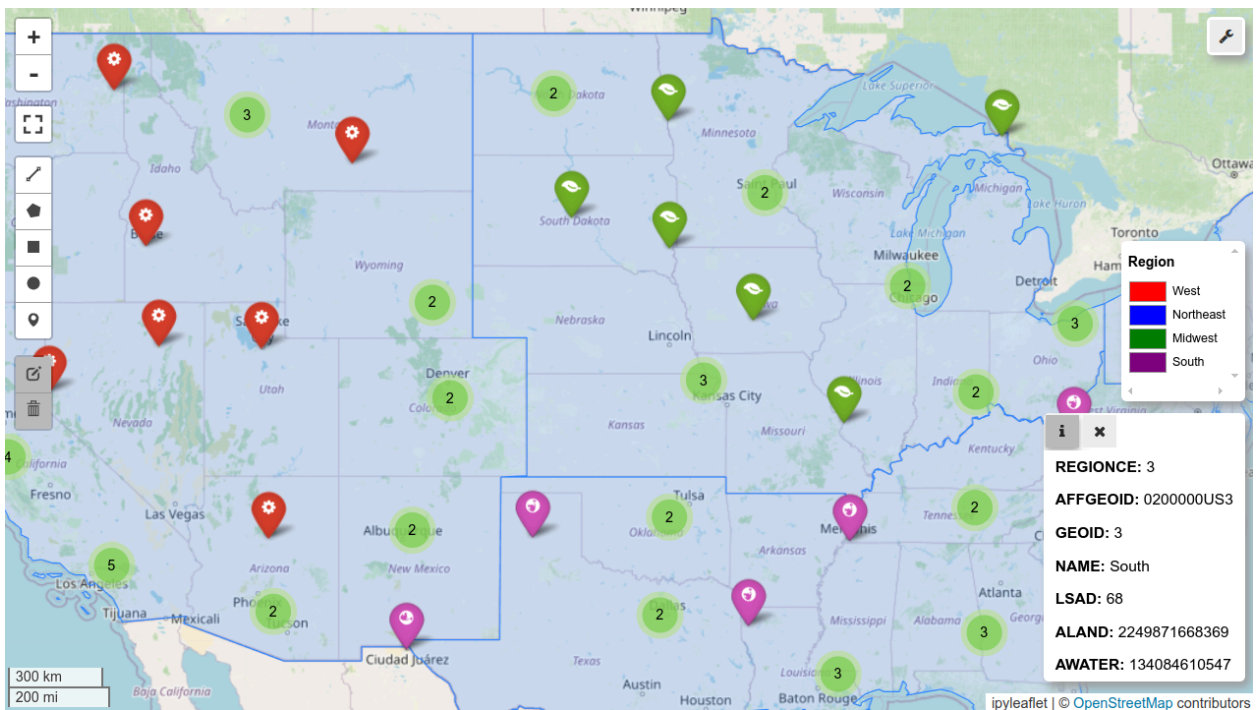


Figure 47: A custom marker cluster.

20.6.2. Visualizing Polylines

Polyline visualization is useful for displaying linear features such as roads or pipelines. In this example, a GeoJSON file containing submarine cable lines is added to the map:

```

m = leafmap.Map(center=[20, 0], zoom=2)
data = "https://github.com/opengeos/datasets/releases/download/vector/cables.
geojson"

```

```
m.add_vector(data, layer_name="Cable lines", info_mode="on_hover")
m
```

20.6.2.1. Customizing Polyline Styles

You can further customize polylines with a style callback function. This example dynamically changes the color and weight of each polyline based on its properties (see [Figure 48](#)):

```
m = leafmap.Map(center=[20, 0], zoom=2)
m.add_basemap("CartoDB.DarkMatter")
data = "https://github.com/opengeos/datasets/releases/download/vector/cables.
geojson"
callback = lambda feat: {"color": feat["properties"]["color"], "weight": 2}
m.add_vector(data, layer_name="Cable lines", style_callback=callback)
m
```

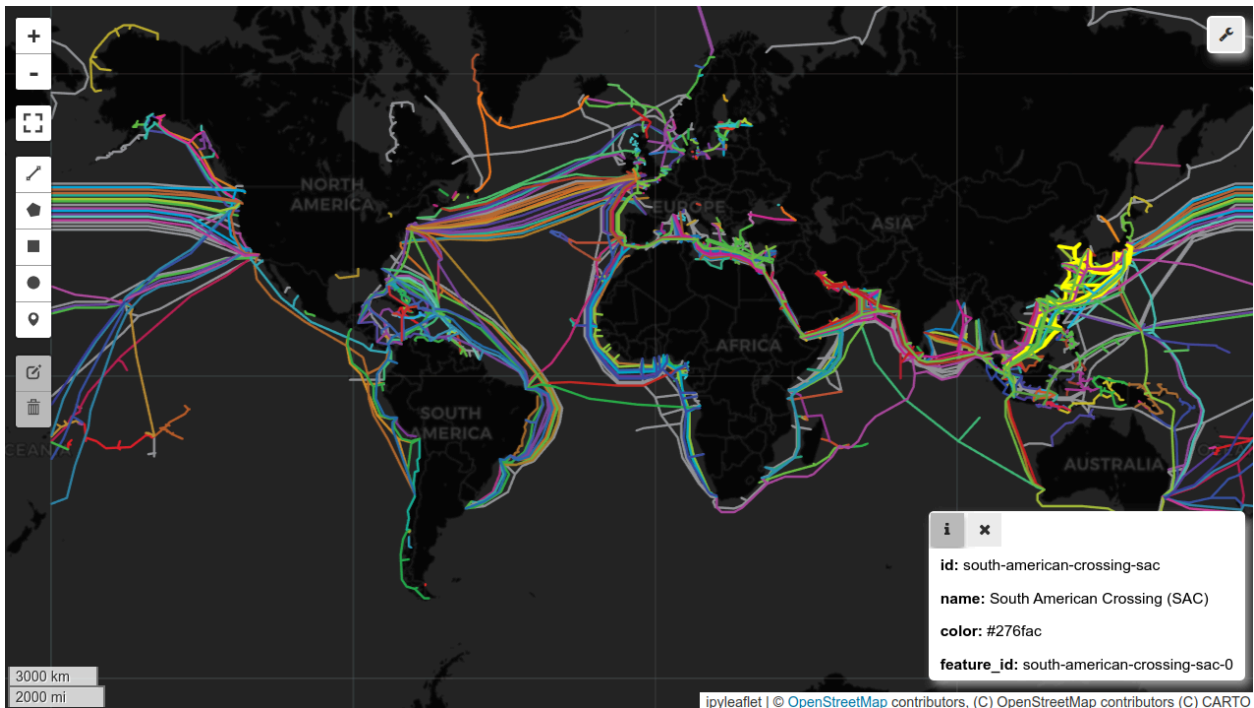


Figure 48: Visualizing undersea cables with leafmap.

20.6.3. Visualizing Polygons

To visualize polygon features, you can use the `Map.add_vector()` method. In this example, a GeoJSON file of New York City buildings is added and the map automatically zooms to the layer's extent:

```
m = leafmap.Map()
url = "https://github.com/opengeos/datasets/releases/download/places/nyc_
```

```
buildings.geojson"
m.add_vector(url, layer_name="NYC Buildings", zoom_to_layer=True)
m
```

20.6.4. Visualizing GeoPandas GeoDataFrames

You can directly visualize GeoPandas GeoDataFrames on the map. In this example, a GeoDataFrame containing building footprints from Las Vegas is displayed with custom styling:

```
url = "https://github.com/opengeos/datasets/releases/download/places/las_vegas_
buildings.geojson"
gdf = leafmap.read_vector(url)
gdf.head()
```

You can use the `GeoDataFrame.explore()` method to visualize the data interactively, which utilizes the `folium` library. The example below displays the building footprints interactively:

```
gdf.explore()
```

To display the GeoDataFrame using Leaflet, use the `add_gdf()` method. The following code adds the building footprints to the map (see [Figure 49](#)):

```
m = leafmap.Map()
m.add_basemap("Esri.WorldImagery")
style = {"color": "red", "fillColor": "red", "fillOpacity": 0.1, "weight": 2}
m.add_gdf(gdf, style=style, layer_name="Las Vegas Buildings", zoom_to_layer=True)
m
```




Figure 49: Visualizing buildings with Leafmap.

20.7. Creating Choropleth Maps

Choropleth maps are useful for visualizing data distributions. The `Map.add_data()` method allows you to create choropleth maps from various vector formats. Below is an example that visualizes population data using the “Quantiles” classification scheme (see [Figure 50](#)):

```
m = leafmap.Map()
data = "https://raw.githubusercontent.com/opengeos/leafmap/master/docs/data/
countries.geojson"
m.add_data(
    data, column="POP_EST", scheme="Quantiles", cmap="Blues",
    legend_title="Population"
)
m
```

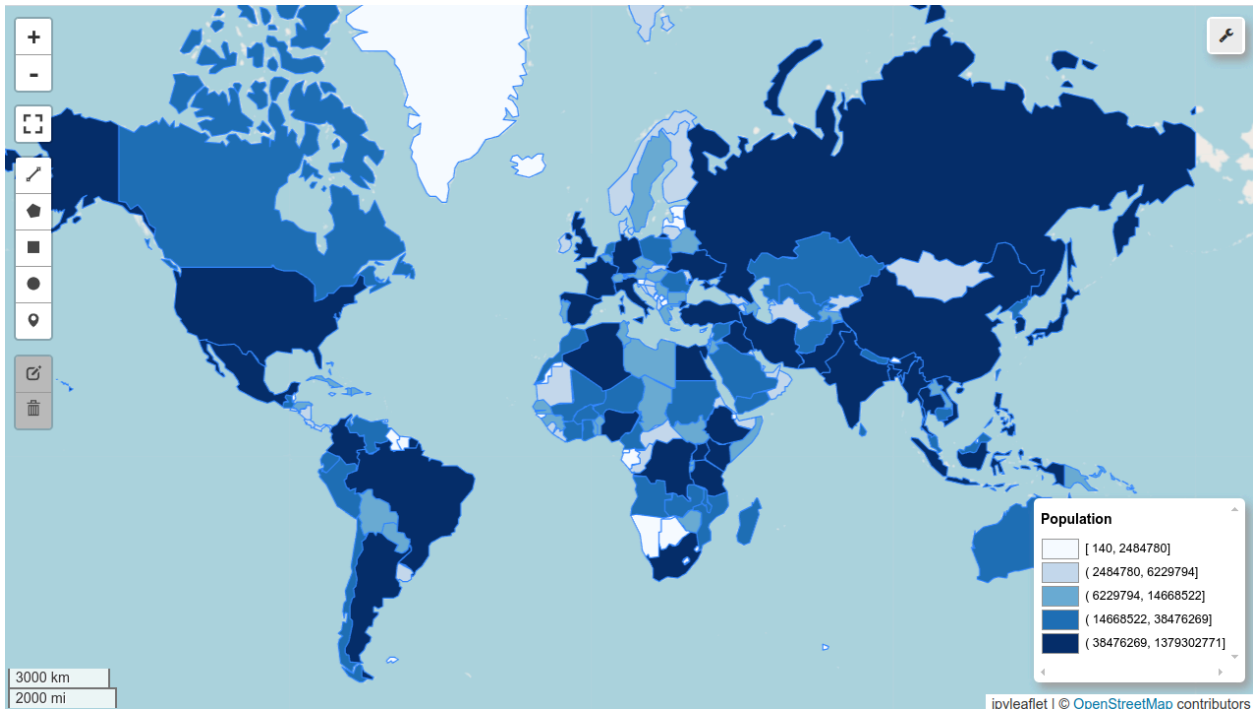


Figure 50: A choropleth map of population data using the “Quantiles” classification scheme.

You can also use different classification schemes like “EqualInterval” for different data visualizations:

```
m = leafmap.Map()
m.add_data(
    data,
    column="POP_EST",
    scheme="EqualInterval",
    cmap="Blues",
    legend_title="Population",
)
m
```

20.8. Visualizing GeoParquet Data

[GeoParquet](https://geoparquet.org)⁶⁷ is a columnar format for geospatial data that allows efficient storage and retrieval. The following example demonstrates how to download, read, and visualize GeoParquet files using GeoPandas and Leafmap.

20.8.1. Loading and Visualizing Point Data

Let’s read a GeoParquet file containing city data from GitHub:

⁶⁷<https://geoparquet.org>


```
url = "https://opengeos.org/data/duckdb/cities.parquet"
gdf = leafmap.read_vector(url)
gdf.head()
```

You can use `GeoDataFrame.explore()` to visualize the data interactively (see [Figure 51](#)).

```
gdf.explore()
```



Figure 51: Visualizing a GeoParquet file with leafmap.

Alternatively, you can add the data to a Leafmap interactive map and plot the points by specifying their latitude and longitude:

```
m = leafmap.Map()
m.add_points_from_xy(gdf, x="longitude", y="latitude")
m
```

20.8.2. Visualizing Polygon Data

For polygon data, such as wetlands, you can follow a similar process. Start by downloading the GeoParquet file containing wetland polygons:

```
url = "https://data.source.coop/giswqs/nwi/wetlands/DC_Wetlands.parquet"
gdf = leafmap.read_vector(url)
gdf.head()
```

You can visualize the polygon data directly using `explore()` with folium:

```
gdf.explore()
```

For visualization with Leafmap, use the following code to add the polygons to the map with a satellite basemap (see [Figure 52](#)).

```
m = leafmap.Map()
m.add_basemap("Esri.WorldImagery", show=False)
m.add_nwi(gdf, col_name="WETLAND_TYPE", zoom_to_layer=True)
m
```

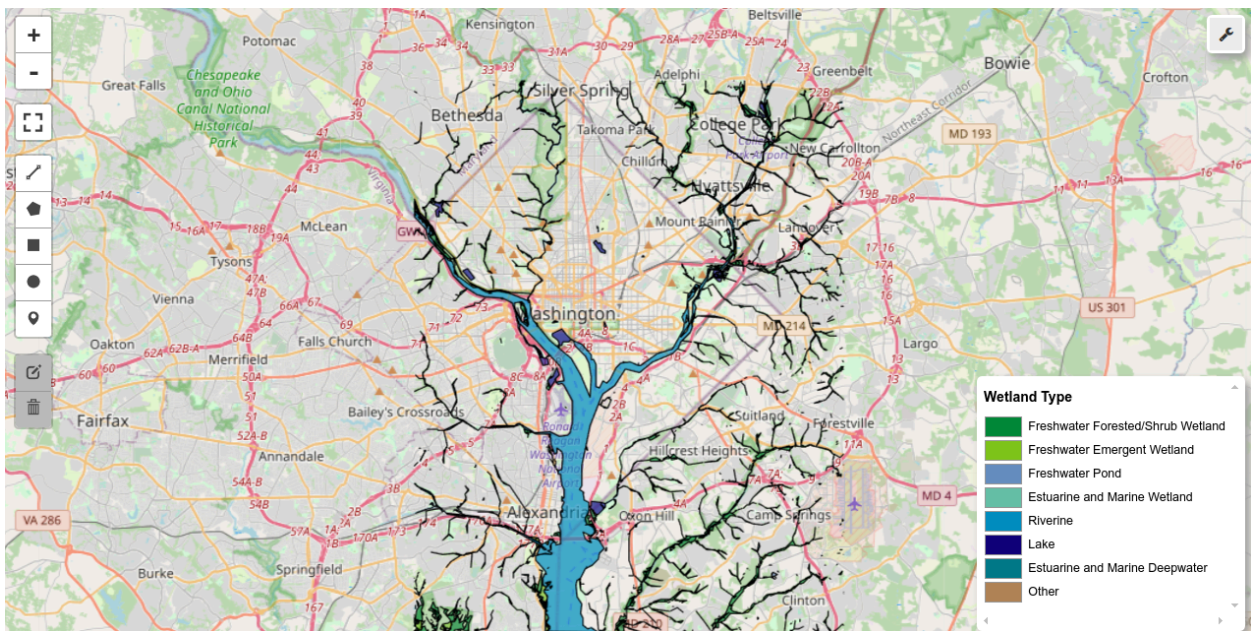


Figure 52: Visualizing the National Wetlands Inventory (NWI) with leafmap.

20.9. Visualizing PMTiles

[PMTiles](#)⁶⁸ is a single-file archive format for tiled data that enables efficient, serverless map applications. PMTiles archives can be hosted on platforms like S3 and allow low-cost, scalable map hosting without the need for custom servers.

20.9.1. Retrieving Metadata from PMTiles

To visualize PMTiles data, first retrieve metadata such as available layers and bounds. Here's an example using a PMTiles archive of Florence, Italy:

```
url = "https://opengeos.org/data/pmtiles/protomaps_firenze.pmtiles"
metadata = leafmap.pmtiles_metadata(url)
```

⁶⁸<https://github.com/protomaps/PMTiles>

```
print(f"layer names: {metadata['layer_names']}")
print(f"bounds: {metadata['bounds']}")
```

20.9.2. Visualizing PMTiles Data

You can add PMTiles layers to a leafmap map by specifying the tile source and defining a custom style. The style specifies how different vector layers, such as buildings and roads, should be rendered:

```
m = leafmap.Map()

style = {
    "version": 8,
    "sources": {
        "example_source": {
            "type": "vector",
            "url": "pmtiles://" + url,
            "attribution": "PMTiles",
        }
    },
    "layers": [
        {
            "id": "buildings",
            "source": "example_source",
            "source-layer": "landuse",
            "type": "fill",
            "paint": {"fill-color": "steelblue"},
        },
        {
            "id": "roads",
            "source": "example_source",
            "source-layer": "roads",
            "type": "line",
            "paint": {"line-color": "black"},
        },
    ],
}

# style = leafmap.pmtiles_style(url) # Use default style
m.add_pmtiles(
    url, name="PMTiles", style=style, overlay=True, show=True, zoom_to_layer=True
)
m
```

20.9.3. Visualizing Open Buildings Data with PMTiles

You can also visualize large datasets like the [Google-Microsoft Open Buildings data](https://source.coop/repositories/vida/google-microsoft-open-buildings/description)⁶⁹ hosted on Source Cooperative. First, check the PMTiles metadata for the building footprints layer:

```
url = "https://data.source.coop/vida/google-microsoft-open-buildings/pmtiles/go_ms_building_footprints.pmtiles"
metadata = leafmap.pmtiles_metadata(url)
print(f"layer names: {metadata['layer_names']}")
print(f"bounds: {metadata['bounds']}")
```

```
layer names: ['building_footprints']
bounds: [-160.221701, -55.9756776, 166.709685, 74.7731168]
```

Now, visualize the building footprints using a custom style for the fill color and opacity:

```
m = leafmap.Map(center=[20, 0], zoom=2)
m.add_basemap("CartoDB.DarkMatter")
m.add_basemap("Esri.WorldImagery", show=False)

style = {
    "version": 8,
    "sources": {
        "example_source": {
            "type": "vector",
            "url": "pmtiles://" + url,
            "attribution": "PMTiles",
        }
    },
    "layers": [
        {
            "id": "buildings",
            "source": "example_source",
            "source-layer": "building_footprints",
            "type": "fill",
            "paint": {"fill-color": "#3388ff", "fill-opacity": 0.5},
        }
    ],
}

m.add_pmtiles(
    url, name="Buildings", style=style, overlay=True, show=True,
    zoom_to_layer=False
)
m
```

⁶⁹<https://source.coop/repositories/vida/google-microsoft-open-buildings/description>

20.9.4. Visualizing Overture Maps Data

[Overture Maps Foundation](https://labs.overturemaps.org)⁷⁰ provides open-source, high-quality basemaps for web mapping applications. You can visualize Overture Maps data using PMTiles archives. The following example demonstrates how to visualize the building footprints layer from the Overture Maps. For more information, visit the [Overture Maps documentation](https://docs.overturemaps.org)⁷¹.

The Overture Maps data are being updated regularly. You can get the latest release of the Overture Maps data using the following code:

```
release = leafmap.get_overture_latest_release()
release
```

At the time of writing, the latest release of the Overture Maps data is 2025-05-21. You can check the historical releases of the Overture Maps data by visiting <https://labs.overturemaps.org/data/releases.json>.

Next, we can visualize the building footprints layer from the Overture Maps (see [Figure 53](#)).

```
theme = "buildings"
url = f"https://overturemaps-tiles-us-west-2-beta.s3.amazonaws.com/{release}/{theme}.pmtiles"
```

```
style = {
    "version": 8,
    "sources": {
        "example_source": {
            "type": "vector",
            "url": "pmtiles://" + url,
            "attribution": "PMTiles",
        }
    },
    "layers": [
        {
            "id": "Building",
            "source": "example_source",
            "source-layer": "building",
            "type": "fill",
            "paint": {
                "fill-color": "#ffff00",
                "fill-opacity": 0.7,
                "fill-outline-color": "#ff0000",
            },
        },
    ],
}
```

⁷⁰<https://overturemaps.org>

⁷¹<https://docs.overturemaps.org>


```
m = leafmap.Map(center=[47.65350739, -117.59664999], zoom=16)
m.add_basemap("Satellite")
m.add_pmtiles(url, style=style, layer_name="Buildings", zoom_to_layer=False)
m
```



Figure 53: Visualizing the building footprints layer from the Overture Maps.

20.10. Visualizing Raster Data

Leafmap supports various raster data formats, including GeoTIFF, Cloud Optimized GeoTIFF (COG), SpatioTemporal Asset Catalog (STAC), and others. This section demonstrates how to visualize raster data using Leafmap.

20.10.1. Visualizing Cloud Optimized GeoTIFFs (COGs)

A Cloud Optimized GeoTIFF ([COG](https://coge.org)⁷²) is a regular GeoTIFF file, aimed at being hosted on a HTTP file server, with an internal organization that enables more efficient workflows on the cloud. It does this by leveraging the ability of clients issuing HTTP GET range requests to ask for just the parts of a file they need.

20.10.1.1. Adding a Cloud Optimized GeoTIFF (COG)

You can load remote COGs directly from a URL. In this example, we load pre-event imagery of the 2020 California fire:

```
m = leafmap.Map(center=[39.494897, -108.507278], zoom=10)
url = "https://opendata.digitalglobe.com/events/california-fire-2020/pre-event/2018-02-16/pine-gulch-fire20/1030010076004E00.tif"
```

⁷²<https://coge.org>

```
m.add_cog_layer(url1, name="Fire (pre-event)")
m
```

Under the hood, the `add_cog_layer()` method uses the [TiTiler](https://developmentseed.org/titiler)⁷³ endpoint to serve COGs as map tiles. The method also supports custom styling and visualization parameters. Please refer to the [TiTiler documentation](https://developmentseed.org/titiler/endpoints/cog/#api)⁷⁴ for more information.

Please note that the `add_cog_layer()` method requires an internet connection to fetch the COG tiles from the TiTiler endpoint. If you need to work offline, you can download the COG and load it as a local GeoTIFF file using the `Map.add_raster()` method covered in the next section.

To show the image metadata, use the `cog_info()` function:

```
leafmap.cog_info(url1)
```

To check the available bands, use the `cog_bands()` function:

```
leafmap.cog_bands(url1)
```

To get the band statistics, use the `cog_stats()` function:

```
stats = leafmap.cog_stats(url1)
# stats
```

20.10.1.2. Adding Multiple COGs

You can visualize and compare multiple COGs by adding them to the same map. Below, we add post-event imagery to the map:

```
url2 = "https://opendata.digitalglobe.com/events/california-fire-2020/post-event/2020-08-14/pine-gulch-fire20/10300100AAC8DD00.tif"
m.add_cog_layer(url2, name="Fire (post-event)")
m
```

20.10.1.3. Creating a Split Map for Comparison

Leafmap also provides a convenient way to compare two COGs side by side using a split map. In this example, we create a map comparing pre-event and post-event fire imagery (see [Figure 54](#)).

```
m = leafmap.Map(center=[39.494897, -108.507278], zoom=10)
m.split_map(
    left_layer=url1, right_layer=url2, left_label="Pre-event", right_label="Post-
event"
```

⁷³<https://developmentseed.org/titiler>

⁷⁴<https://developmentseed.org/titiler/endpoints/cog/#api>

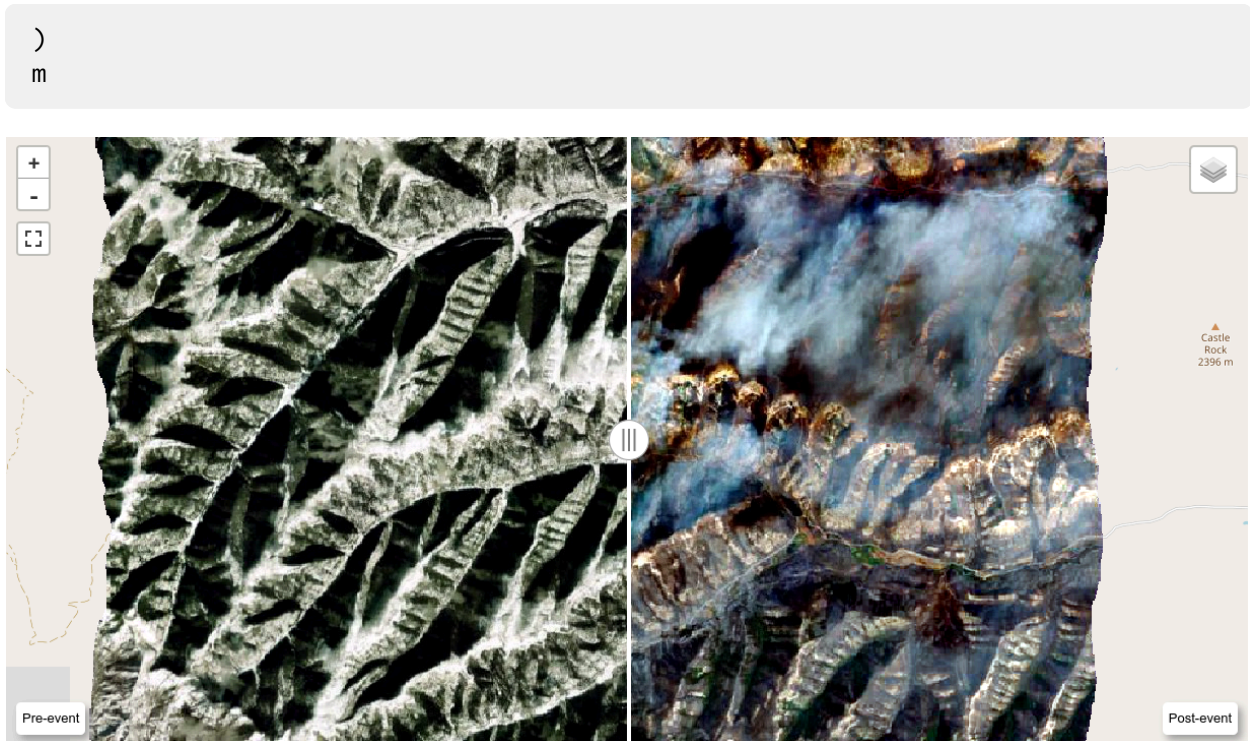


Figure 54: Visualizing the pre-event and post-event imagery of the 2020 California fire with leafmap. Here is another example comparing two COGs using a split map (Figure 54).

```
m = leafmap.Map(center=[47.653149, -117.59825], zoom=16)
m.add_basemap("Satellite")
image1 = "https://github.com/opengeos/datasets/releases/download/places/wa_
building_image.tif"
image2 = "https://github.com/opengeos/datasets/releases/download/places/wa_
building_masks.tif"
m.split_map(
    image2,
    image1,
    left_label="Building Masks",
    right_label="Aerial Imagery",
    left_args={"colormap_name": "tab20", "nodata": 0, "opacity": 0.7},
)
m
```

20.10.1.4. Using a Custom Colormap

You can apply a custom colormap to raster data for better visualization. The example below shows how to visualize the [US National Land Cover Database \(NLCD\)](https://www.mrlc.gov/data/nlcd-2021-land-cover-conus)⁷⁵ data with a custom colormap (Figure 55):

⁷⁵<https://www.mrlc.gov/data/nlcd-2021-land-cover-conus>


```

url = "https://github.com/opengeos/datasets/releases/download/raster/nlcd_2021_
land_cover_30m.tif"
colormap = {
    "11": "#466b9f",
    "12": "#d1def8",
    "21": "#dec5c5",
    "22": "#d99282",
    "23": "#eb0000",
    "24": "#ab0000",
    "31": "#b3ac9f",
    "41": "#68ab5f",
    "42": "#1c5f2c",
    "43": "#b5c58f",
    "51": "#af963c",
    "52": "#ccb879",
    "71": "#dfdfc2",
    "72": "#d1d182",
    "73": "#a3cc51",
    "74": "#82ba9e",
    "81": "#dcd939",
    "82": "#ab6c28",
    "90": "#b8d9eb",
    "95": "#6c9fb8",
}
}
m = leafmap.Map(center=[40, -100], zoom=4, height="650px")
m.add_basemap("Esri.WorldImagery")
m.add_cog_layer(url, colormap=colormap, name="NLCD Land Cover", nodata=0)
m.add_legend(title="NLCD Land Cover Type", builtin_legend="NLCD")
m.add_layer_manager()
m

```

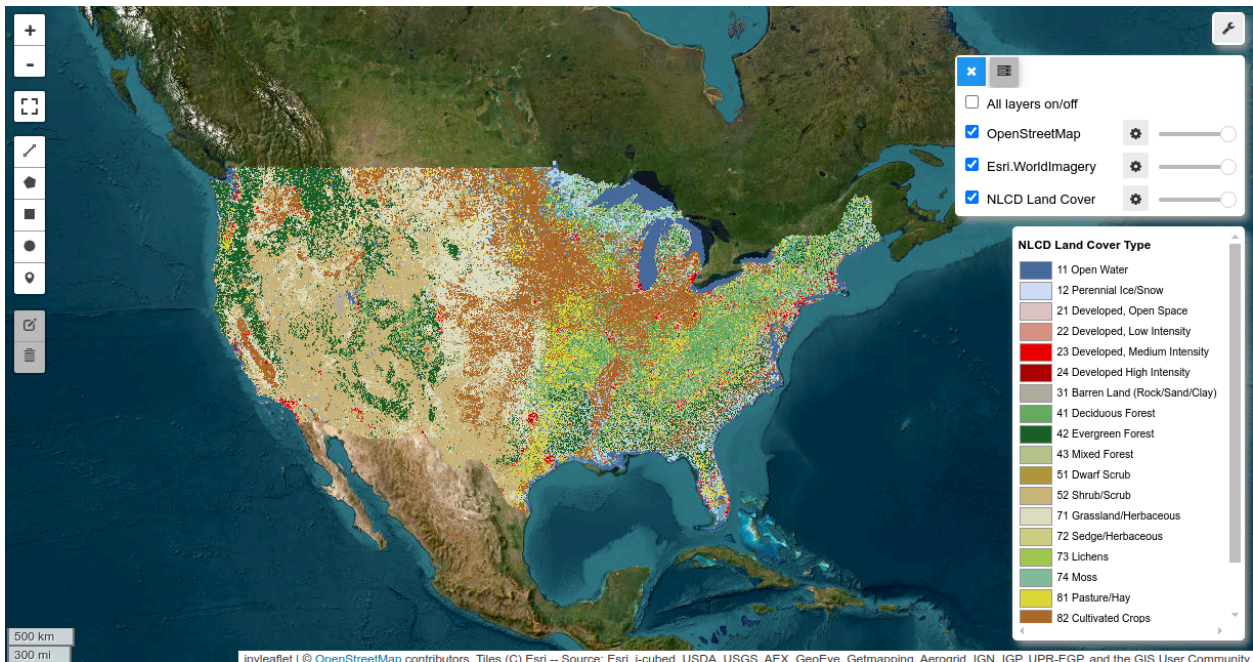


Figure 55: Visualizing the US National Land Cover Database (NLCD) data with a custom colormap.

20.10.2. Visualizing Local Raster Datasets

Leafmap supports visualizing local GeoTIFF datasets as well. In this example, we download a sample Digital Elevation Model (DEM) dataset and visualize it.

20.10.2.1. Downloading and Visualizing a Local Raster

Start by downloading a sample DEM GeoTIFF file:

```
dem_url = "https://github.com/opengeos/datasets/releases/download/raster/dem_90m.tif"
filename = "dem_90m.tif"
leafmap.download_file(dem_url, filename, quiet=True)
```

Next, visualize the raster using the `Map.add_raster()` method with a “terrain” colormap to highlight elevation differences (Figure 56):

```
m = leafmap.Map()
m.add_raster(filename, colormap="terrain", layer_name="DEM")
m
```

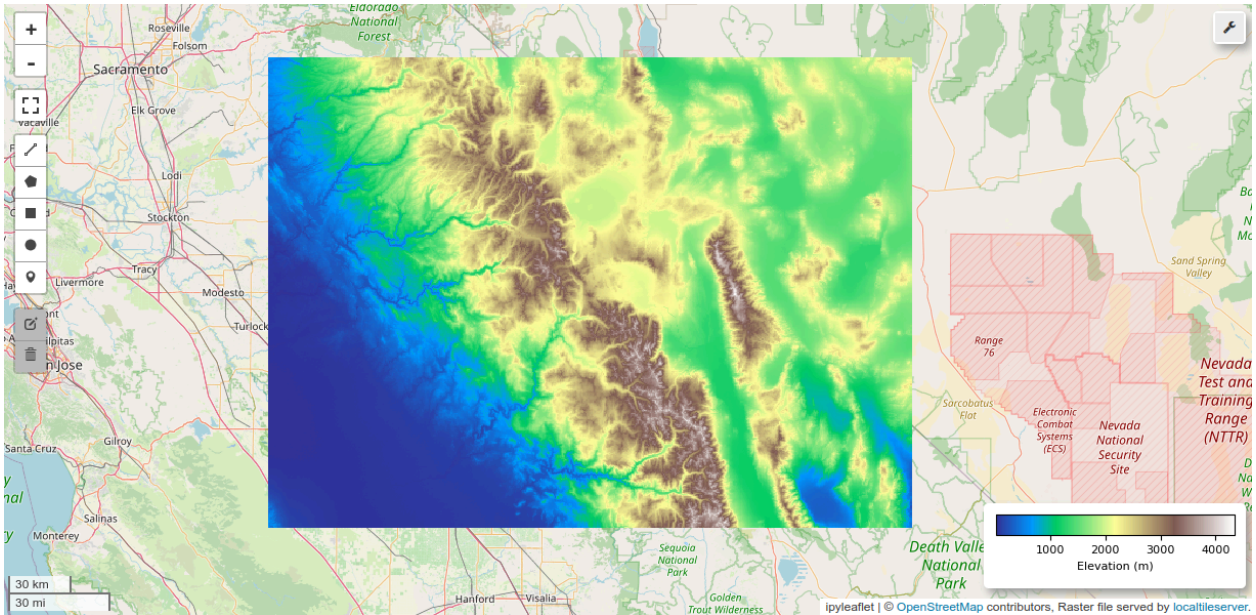


Figure 56: Visualizing the DEM data with a terrain colormap.

You can also check the minimum and maximum values of the raster:

```
leafmap.image_min_max(filename)
```

Optionally, add a colormap legend to indicate the range of elevation values:

```
m.add_colormap(cmap="terrain", vmin=15, vmax=4338, label="Elevation (m)")
```

Keep in mind that the `add_raster()` method works for both local and remote GeoTIFF files. You can use it to visualize COGs available online or local GeoTIFF files stored on your computer. The example below demonstrates how to visualize the same COG from the previous section as a remote file:

20.10.2.2. Visualizing a Multi-Band Raster

Multi-band rasters, such as satellite images, can also be visualized. The following example loads a multi-band Landsat image and displays it as an RGB composite.

```
import leafmap
```

```
landsat_url = "https://github.com/opengeos/datasets/releases/download/raster/cog.tif"
filename = "cog.tif"
leafmap.download_file(landsat_url, filename, quiet=True)
```

```
m = leafmap.Map()
m.add_raster(filename, indexes=[4, 3, 2], layer_name="RGB")
m
```

20.10.2.3. Inspecting Pixel Values

To inspect pixel values interactively, use the `Map.add("inspector")` method to add an inspector control to the map. The inspector control displays pixel values when you click on the map (Figure 57).

```
m = leafmap.Map(center=[53.407089, 6.875480], zoom=13)
m.add_raster(filename, indexes=[4, 3, 2], layer_name="RGB")
m.add("inspector")
m
```



Figure 57: Inspecting pixel values with leafmap.

20.10.3. Visualizing SpatioTemporal Asset Catalog (STAC) Data

The [STAC specification](https://stacspec.org)⁷⁶ provides a standardized way to describe geospatial information, enabling easier discovery and use. In this section, we will visualize STAC data using Leafmap.

20.10.3.1. Exploring STAC Bands

To begin, retrieve the available bands for a STAC item. This example uses the [SPOT Orthoimages of Canada](https://stacindex.org/catalogs/spot-orthoimages-canada-2005)⁷⁷:

⁷⁶<https://stacspec.org>

⁷⁷<https://stacindex.org/catalogs/spot-orthoimages-canada-2005>


```
url = "https://canada-spot-ortho.s3.amazonaws.com/canada_spot_orthoimages/canada_
spot5_orthoimages/S5_2007/S5_11055_6057_20070622/S5_11055_6057_20070622.json"
leafmap.stac_bands(url)
```

```
['pan', 'B1', 'B2', 'B3', 'B4']
```

20.10.3.2. Adding STAC Layers to the Map

Once you have explored the available bands, you can add them to your map. In this example, we visualize both the panchromatic band and false-color composite of the SPOT Orthoimage ([Figure 58](#)).

```
m = leafmap.Map(center=[60.95410, -110.90184], zoom=10)
m.add_stac_layer(url, bands=["pan"], name="Panchromatic")
m.add_stac_layer(url, bands=["B3", "B2", "B1"], name="False color")
m
```

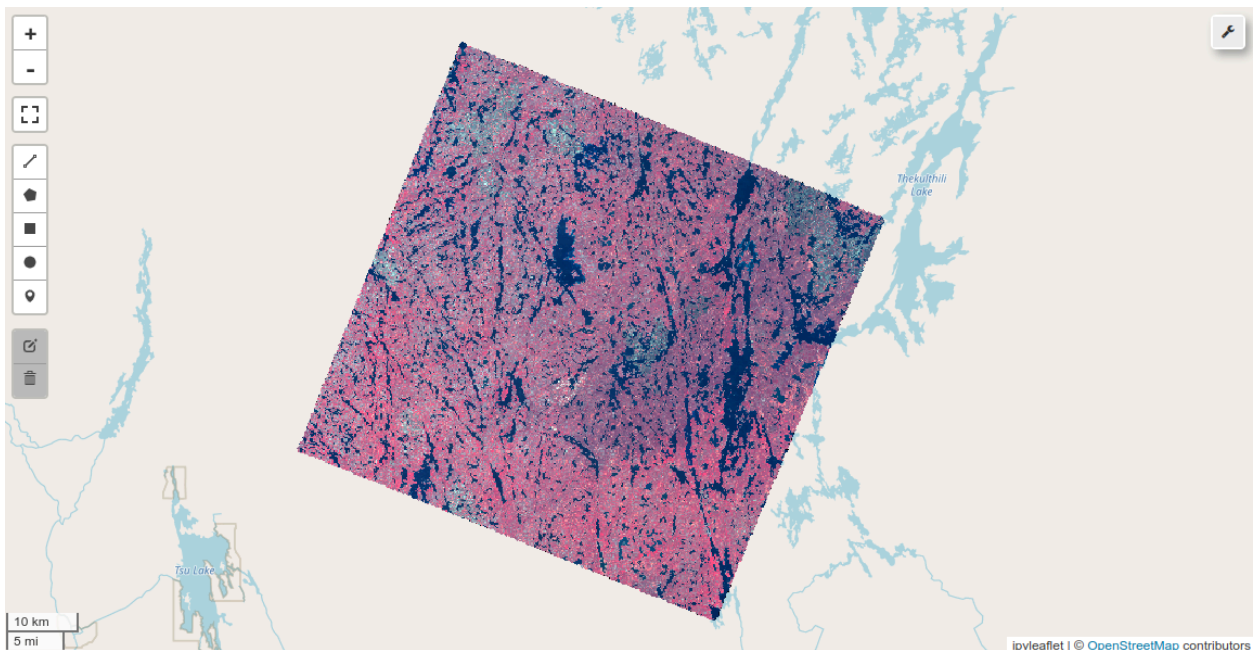


Figure 58: Visualizing STAC data with leafmap.

20.11. Accessing and Visualizing Maxar Open Data

The **Maxar Open Data Program** provides high-resolution satellite imagery in the aftermath of natural disasters and humanitarian crises. Unlike continuous monitoring programs, Maxar's initiative is event-driven—activated during emergencies such as hurricanes, wildfires, earthquakes, and conflicts. By releasing timely, publicly available imagery, Maxar empowers responders, analysts, and volunteers with actionable insights for damage assessment, response coordination, and recovery planning. More

information about Maxar Open Data is available at <https://registry.opendata.aws/maxar-open-data>⁷⁸.

20.11.1. Discovering Available Disaster Events

The first step is to explore what disaster events are available in the Maxar Open Data catalog. Each collection in the catalog represents a single disaster event with associated satellite imagery:

```
leafmap.maxar_collections()
```

This function retrieves all available collections from the Maxar Open Data STAC (SpatioTemporal Asset Catalog) catalog. The output will show you a list of disaster events, each with a unique collection ID that we can use to access the specific imagery data.

20.11.2. Selecting a Disaster Event

For this example, we'll focus on the devastating earthquake that struck Turkey and Syria in February 2023. The collection ID for this event is `Kahramanmaras-turkey-earthquake-23`. We can retrieve the geographic footprints of all available satellite images for this event from the [Maxar Open Data GitHub repository](#), which provides both GeoJSON and TSV formats:

```
collection = "Kahramanmaras-turkey-earthquake-23"
url = leafmap.maxar_collection_url(collection, dtype="geojson")
url
```

This function generates a URL pointing to the GeoJSON file containing the spatial footprints and metadata for all satellite images related to the Turkey earthquake event. The GeoJSON format is particularly useful because it includes both the geographic boundaries of each image and associated metadata.

Now let's load this data and explore what's available:

```
gdf = leafmap.read_vector(url)
print(f"Total number of images: {len(gdf)}")
gdf.head()
```

Here we're using GeoPandas to read the remote GeoJSON file directly from the URL. The `gdf.head()` command shows us the first few rows of the dataset, revealing important metadata columns such as:

- **geometry**: The spatial footprint of each satellite image
- **catalog_id**: A unique identifier for grouping related images
- **quadkey**: Microsoft's quadkey system for tile identification
- **timestamp**: When the image was captured
- **visual**: Direct download link to the satellite image

20.11.3. Visualizing Image Footprints

Now let's create an interactive map to visualize where all these satellite images are located geographically:

⁷⁸<https://registry.opendata.aws/maxar-open-data>

```
m = leafmap.Map()
m.add_gdf(gdf, layer_name="Footprints", zoom_to_layer=True)
m
```

This creates an interactive map centered on the earthquake region, with blue polygons showing the spatial coverage of each satellite image. You can zoom in, pan around, and click on individual footprints to see their metadata. This visualization helps us understand the geographic extent of the available imagery and identify areas with dense coverage.

20.11.4. Temporal Analysis: Before and After the Earthquake

The earthquake struck on February 6, 2023, making temporal analysis crucial for damage assessment. Let's separate the imagery into pre-event and post-event datasets using the earthquake date as our temporal boundary.

First, let's get all images captured before the earthquake:

```
pre_gdf = leafmap.maxar_search(collection, end_date="2023-02-06")
print(f"Total number of pre-event images: {len(pre_gdf)}")
pre_gdf.head()
```

The `leafmap.maxar_search()` function allows us to filter the imagery collection by date. By setting `end_date="2023-02-06"`, we retrieve only images captured before the earthquake occurred. These pre-event images serve as our baseline for comparison.

Now let's get all images captured after the earthquake:

```
post_gdf = leafmap.maxar_search(collection, start_date="2023-02-06")
print(f"Total number of post-event images: {len(post_gdf)}")
post_gdf.head()
```

By setting `start_date="2023-02-06"`, we retrieve all images captured from the earthquake date onwards. These post-event images will show the immediate aftermath and damage caused by the earthquake.

20.11.5. Comparing Pre-Event and Post-Event Coverage

Let's create a comparative visualization showing both pre-event and post-event image footprints on the same map ([Figure 59](#)):

```
m = leafmap.Map()
pre_style = {"color": "red", "fillColor": "red", "opacity": 1, "fillOpacity": 0.5}
m.add_gdf(
    pre_gdf,
    layer_name="Pre-event",
    style=pre_style,
    info_mode="on_click",
```

```

        zoom_to_layer=True,
    )
    m.add_gdf(post_gdf, layer_name="Post-event", info_mode="on_click",
              zoom_to_layer=True)
    m

```

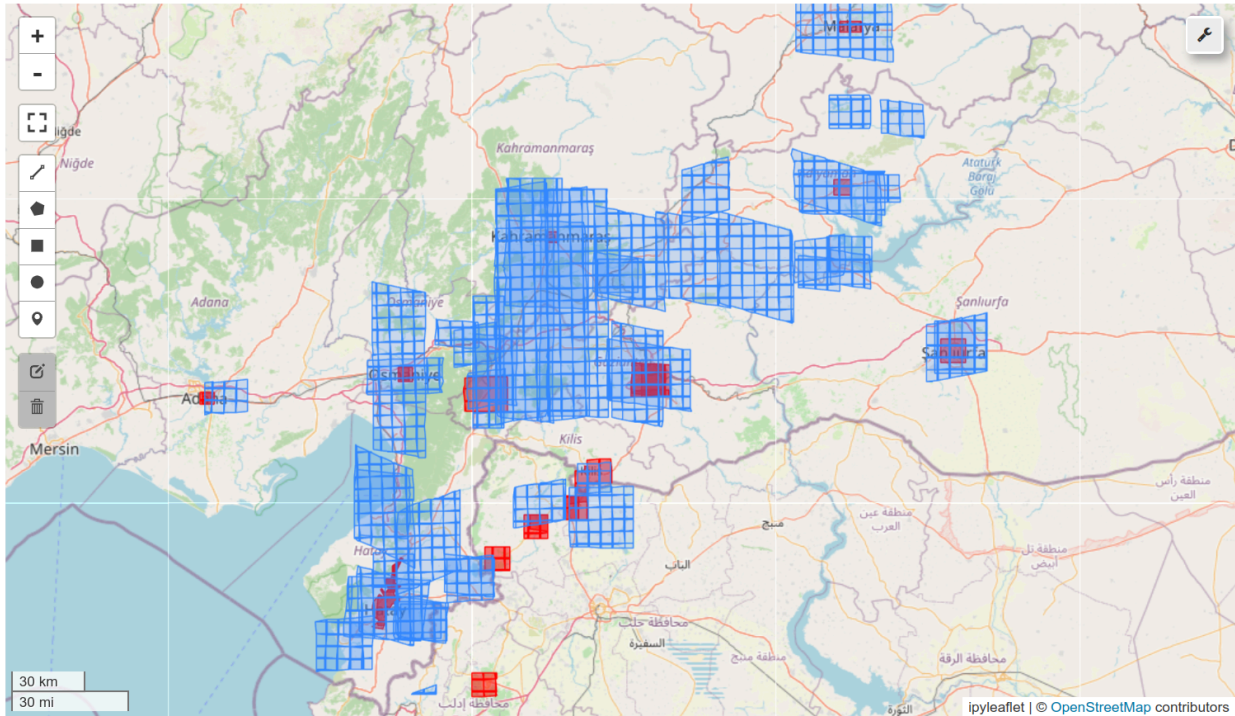


Figure 59: The interactive map shows the spatial coverage of pre-event (red) and post-event (blue) satellite images for the Turkey earthquake.

In this visualization, red polygons represent pre-earthquake imagery while blue polygons show post-earthquake coverage. The `info_mode="on_click"` parameter enables interactive information popups when you click on any footprint. You can toggle layers on/off using the layer control panel, and the different colors help distinguish the temporal coverage patterns.

20.11.6. Selecting a Region of Interest

To focus our analysis on a specific area, we can define a region of interest (ROI). You can either draw a polygon on the map using the drawing tools, or we'll use predefined coordinates for a particularly affected area:

```

bbox = m.user_roi_bounds()
if bbox is None:
    bbox = [36.8715, 37.5497, 36.9814, 37.6019]

```

The `m.user_roi_bounds()` function attempts to get the bounding box coordinates from any region you've drawn on the map. If no region is drawn, we fall back to predefined coordinates that cover a

significantly impacted area near Kahramanmaraş. The bbox format follows [min_longitude, min_latitude, max_longitude, max_latitude] convention.

20.11.7. Searching Within the Region of Interest

Now let's search for satellite images that specifically cover our region of interest, filtering by both geographic bounds and temporal constraints.

First, let's find pre-earthquake images within our ROI:

```
pre_event = leafmap.maxar_search(collection, bbox=bbox, end_date="2023-02-06")
pre_event.head()
```

This search combines spatial filtering (using the bounding box) with temporal filtering (images before February 6, 2023). The result shows us only images that both intersect our geographic area of interest and were captured before the earthquake.

Similarly, let's find post-earthquake images within the same area:

```
post_event = leafmap.maxar_search(collection, bbox=bbox, start_date="2023-02-06")
post_event.head()
```

This gives us the corresponding post-earthquake imagery for the same geographic region, enabling direct before-and-after comparison.

20.11.8. Preparing Images for Visualization

Maxar organizes satellite images into tiles, where each tile can contain multiple individual images identified by unique quadkeys. Let's extract the tile identifiers we need for visualization:

```
pre_tile = pre_event["catalog_id"].values[0]
pre_tile
```

This gets the catalog ID for the first pre-event tile in our search results. The catalog ID serves as a unique identifier for a collection of related satellite images covering the same geographic area.

```
post_tile = post_event["catalog_id"].values[0]
post_tile
```

Similarly, this extracts the catalog ID for the post-event tile. Having both pre- and post-event tile IDs allows us to create comparative visualizations.

20.11.9. Creating Web-Optimized Tile Services

To display these high-resolution satellite images efficiently in a web browser, we need to convert them to MosaicJSON format, which enables dynamic tiling and streaming:

```
pre_stac = leafmap.maxar_tile_url(collection, pre_tile, dtype="json")
pre_stac
```

This generates a MosaicJSON URL for the pre-event tile. MosaicJSON is a specification that allows multiple raster files to be presented as a single web-optimized layer, enabling efficient visualization of large satellite imagery datasets.

```
post_stac = leafmap.maxar_tile_url(collection, post_tile, dtype="json")
post_stac
```

Similarly, this creates the MosaicJSON URL for the post-event imagery. These URLs point to tile services that can stream the imagery directly to our interactive map.

20.11.10. Creating a Split-Map Comparison

Now comes the powerful part—creating a side-by-side comparison to visualize the earthquake’s impact ([Figure 60](#)).

```
m = leafmap.Map()
m.split_map(
    left_layer=pre_stac,
    right_layer=post_stac,
    left_label="Pre-event",
    right_label="Post-event",
)
m.set_center(36.9265, 37.5762, 16)
m
```

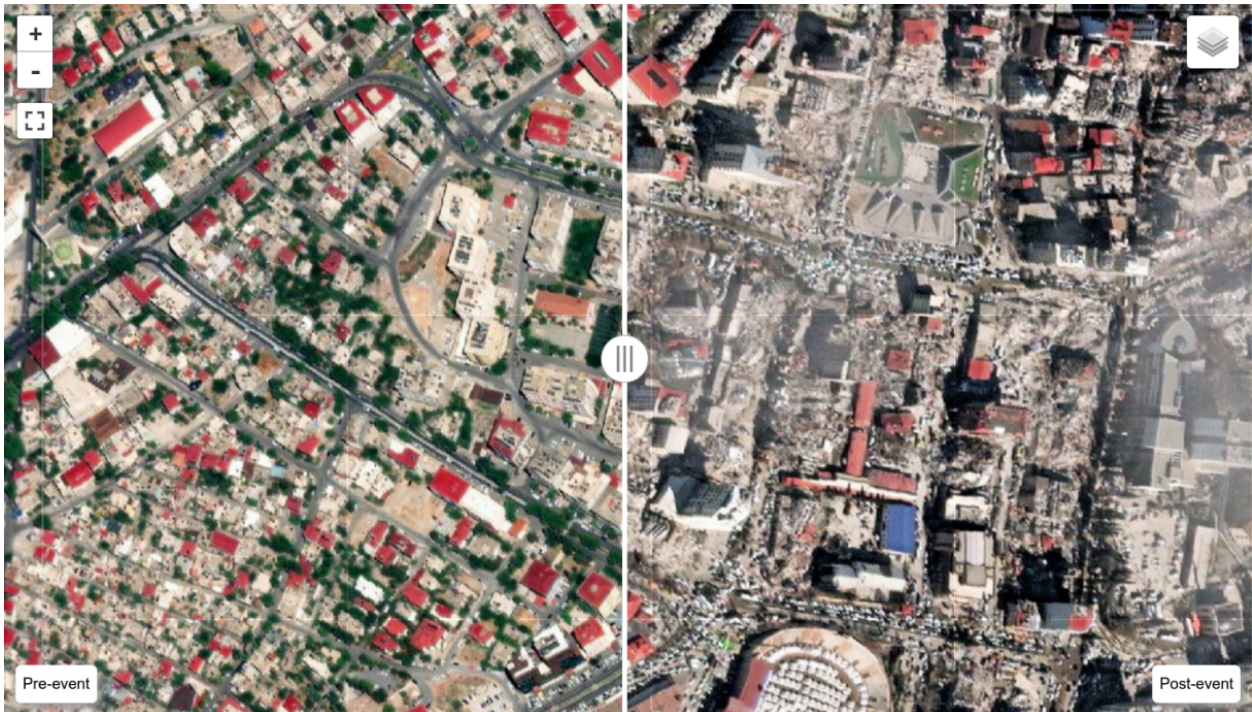


Figure 60: The split map shows the pre-event (left) and post-event (right) satellite images for the Turkey earthquake.

This creates an interactive split-screen map where you can:

- **Left side:** View the area before the earthquake
- **Right side:** View the same area after the earthquake
- **Divider:** Drag the vertical divider left or right to compare different parts of the scene
- **Synchronization:** Both sides pan and zoom together, maintaining perfect geographic alignment

20.11.11. Downloading Images for Offline Analysis

If you need to perform detailed analysis or store images locally, you can download the original high-resolution imagery:

```
pre_images = pre_event["visual"].tolist()
post_images = post_event["visual"].tolist()
```

This extracts the direct download URLs from our filtered datasets. The “visual” column contains links to the processed, analysis-ready satellite images in RGB format that are optimized for visual interpretation.

Now let’s download the pre-event images:

```
leafmap.maxar_download(pre_images)
```

The `leafmap.maxar_download()` function handles the download process, saving images to your local directory with organized naming conventions. These high-resolution images can then be used for:

- Detailed damage assessment
- Machine learning training datasets
- Offline mapping applications
- Scientific research and analysis

```
# leafmap.maxar_download(post_images)
```

There are a lot more post-event images available for the Turkey earthquake. It may take a while to download all the images. Uncomment the above cell to download the post-event images if needed.

20.12. Key Takeaways

This chapter has introduced **Leafmap**, a versatile Python library for creating, managing, and analyzing interactive geospatial maps. It covers essential tools for GIS programming, allowing for streamlined workflows in handling spatial data. Leafmap's integration with Jupyter environments enables users to visualize and analyze geospatial data with minimal code.

Key takeaways from this chapter include:

1. **Setting Up Leafmap:** Instructions for installing and importing Leafmap, alongside switching between supported plotting backends.
2. **Interactive Map Customization:** Techniques to create and customize maps, add various basemaps, and control map elements like zoom, scale, and toolbar settings.
3. **Data Layer Management:** Methods for adding/removing layers, customizing basemaps, and incorporating layers like WMS and XYZ tiles for expanded data visualization.
4. **Vector and Raster Data Visualization:** Examples on adding vector data (points, lines, polygons) from formats like GeoJSON and GeoParquet and visualizing raster formats like GeoTIFF, COG, and STAC.
5. **Advanced Data Interactions:** How to use specialized formats and datasets, such as PMTiles, Open Buildings data, and STAC API catalogs, and AWS data integration for remote geospatial data.

This chapter provides the foundational knowledge to effectively use Leafmap in GIS projects, making it an invaluable tool for visualizing and analyzing diverse geospatial datasets within Python.

20.13. Exercises

20.13.1. Exercise 1: Creating an Interactive Map

1. Create an interactive map with search functionality that allows users to search for places and zoom to them. Disable the draw control on the map.

20.13.2. Exercise 2: Adding XYZ and WMS Tile Layers

1. Add a custom XYZ tile layer ([USGS Topographic basemap](https://apps.nationalmap.gov/services/))⁷⁹ using the following URL:

⁷⁹<https://apps.nationalmap.gov/services/>

- URL: `https://basemap.nationalmap.gov/arcgis/rest/services/USGSTopo/MapServer/tile/{z}/{y}/{x}`
2. Add two WMS layers to visualize NAIP imagery and NDVI using a USGS WMS service.
 - URL: `https://imagery.nationalmap.gov/arcgis/services/USGSNAIPImagery/ImageServer/WMServer?`
 - Layer names: `USGSNAIPImagery:FalseColorComposite`, `USGSNAIPImagery:NDVI_Color`

20.13.3. Exercise 3: Adding Map Legends

1. Add the [ESA World Cover](https://esa-worldcover.org/en)⁸⁰ WMS tile layer to the map.
 - URL: `https://services.terrascope.be/wms/v2?`
 - Layer name: `WORLDCOVER_2021_MAP`
2. Add a legend to the map using the leafmap built-in `ESA_WorldCover` legend.

20.13.4. Exercise 4: Creating Marker Clusters

1. Create a marker cluster visualization from a GeoJSON file of building centroids:
 - URL: https://github.com/opengeos/datasets/releases/download/places/wa_building_centroids.geojson
 - Hint: Read the GeoJSON file using GeoPandas and add “latitude” and “longitude” columns to the GeoDataFrame.
2. Create circle markers for each building centroid using the `Map.add_circle_markers_from_xy()` method with the following styling:
 - Radius: 5
 - Outline color: “red”
 - Fill color: “yellow”
 - Fill opacity: 0.8

20.13.5. Exercise 5: Visualizing Vector Data

1. Visualize the building polygons GeoJSON file and style it with:
 - Outline color: “red”
 - No fill color
 - URL: https://github.com/opengeos/datasets/releases/download/places/wa_overture_buildings.geojson
2. Visualize the road polylines GeoJSON file and style it with:
 - Line color: “red”

⁸⁰<https://esa-worldcover.org/en>

- Line width: 2
 - URL: https://github.com/opengeos/datasets/releases/download/places/las_vegas_roads.geojson
3. Create a choropleth map of county areas in the US:
- URL: https://github.com/opengeos/datasets/releases/download/us/us_counties.geojson
 - Column: CENSUSAREA

20.13.6. Exercise 6: Visualizing GeoParquet Data

1. Visualize GeoParquet data of US states:
- URL: https://github.com/opengeos/datasets/releases/download/us/us_states.parquet
 - Style: Outline color of “red” with no fill.

20.13.7. Exercise 7: Visualizing PMTiles

1. Visualize the Overture Maps building dataset using PMTiles:
- URL: <https://overturemaps-tiles-us-west-2-beta.s3.amazonaws.com/2024-09-18/buildings.pmtiles>
 - Style: Blue fill with 0.4 opacity, red outline.

20.13.8. Exercise 8: Visualizing Cloud Optimized GeoTIFFs (COGs)

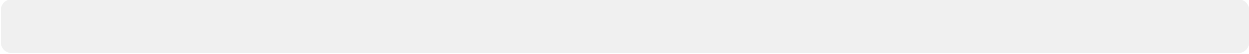
1. Visualize Digital Elevation Model (DEM) data using the following COG file:
- URL: <https://github.com/opengeos/datasets/releases/download/raster/dem.tif>
 - Apply a terrain colormap to show elevation values.

20.13.9. Exercise 9: Visualizing Local Raster Data

1. Visualize the following raster datasets using the Map.add_raster() method:
- Aerial Imagery: https://github.com/opengeos/datasets/releases/download/places/wa_building_image.tif
 - Building Footprints: https://github.com/opengeos/datasets/releases/download/places/wa_building_masks.tif (use the “tab20” colormap and opacity of 0.7)

20.13.10. Exercise 10: Creating a Split Map

1. Create a split map to compare imagery of Libya before and after the 2023 flood event:

- Pre-event imagery: <https://github.com/opengeos/datasets/releases/download/raster/Libya-2023-07-01.tif>
 - Post-event imagery: <https://github.com/opengeos/datasets/releases/download/raster/Libya-2023-09-13.tif>
- 

Chapter 21. Geoprocessing with WhiteboxTools

21.1. Introduction

Geospatial analysis forms the backbone of modern environmental science, urban planning, and natural resource management. Among the many tools available for such analysis, **WhiteboxTools**⁸¹ stands out as a particularly powerful and accessible open-source library. This chapter will guide you through the fundamental concepts and practical applications of WhiteboxTools, focusing on two essential areas of geospatial analysis: watershed analysis and LiDAR data processing.

WhiteboxTools excels in hydrological modeling and terrain analysis, making it an ideal choice for understanding how water moves across landscapes and how we can extract meaningful information from three-dimensional point cloud data. Throughout this chapter, we will work with real-world datasets to demonstrate these concepts, providing you with hands-on experience that you can apply to your own research and projects.

The chapter is organized into two complementary sections. First, we will explore watershed analysis, where you will learn to work with Digital Elevation Models (DEMs) to understand water flow patterns, delineate watershed boundaries, and extract stream networks. This foundation in hydrological analysis is crucial for environmental modeling and water resource management. Second, we will delve into LiDAR data analysis, where you will discover how to process three-dimensional point cloud data to create various elevation models and extract vegetation information.

By combining **WhiteboxTools** with **leafmap**, you will gain the ability to not only perform sophisticated analyses but also visualize and interpret your results effectively. This integration of analysis and visualization is essential for communicating geospatial findings to both technical and non-technical audiences.

21.2. Learning Objectives

By the end of this chapter, you will be able to:

- Install and configure WhiteboxTools and leafmap for geospatial analysis
- Create interactive maps to visualize basemaps and geospatial datasets
- Perform watershed analysis by delineating watersheds, flow directions, and stream networks
- Manipulate and analyze Digital Elevation Models (DEMs) to conduct hydrological modeling
- Process and analyze LiDAR data to generate Digital Surface Models (DSMs), Digital Elevation Models (DEMs), and Canopy Height Models (CHMs)
- Identify and remove data outliers to maintain data quality in geospatial analysis
- Integrate WhiteboxTools with Python workflows to automate geospatial analysis and create reproducible analysis pipelines

21.3. Why Whitebox?

Understanding the context and capabilities of WhiteboxTools will help you appreciate its role in the broader geospatial analysis ecosystem. WhiteboxTools represents a modern approach to geospatial computing, designed from the ground up to be fast, reliable, and easy to integrate into existing workflows.

⁸¹<https://github.com/jblindsay/whitebox-tools>

21.3.1. What is Whitebox?

Whitebox is a comprehensive geospatial data analysis platform that originated from the research efforts at the University of Guelph's Geomorphometry and Hydrogeomatics Research Group (GHRG⁸²), under the direction of Dr. John Lindsay. The platform has evolved into a robust toolkit containing over 550 specialized tools for processing diverse types of geospatial data.

What makes Whitebox particularly appealing is its design philosophy. The developers prioritized performance through extensive use of parallel computing using Rust⁸³, which means that many operations can take advantage of multiple processor cores to complete tasks faster. Additionally, Whitebox is self-contained, meaning it doesn't require you to install and manage complex dependencies like GDAL, which can sometimes be challenging to configure properly across different operating systems.

The platform's architecture also makes it highly scriptable, allowing you to integrate Whitebox functions seamlessly into Python, R, or other programming environments. This flexibility means you can incorporate Whitebox capabilities into larger analytical workflows without having to switch between different software packages.

21.3.2. What can Whitebox do?

Whitebox serves as a comprehensive analytical engine for geospatial and remote sensing tasks. Its strength lies particularly in spatial hydrological analysis, where it provides sophisticated tools for modeling water flow, watershed delineation, and stream network analysis. These capabilities make it invaluable for environmental scientists studying water resources, flood modeling, and landscape hydrology.

The platform also excels in LiDAR data processing, offering specialized tools for handling three-dimensional point cloud data. This includes capabilities for generating various types of elevation models, filtering noise from point clouds, and extracting vegetation metrics from airborne laser scanning data.

Beyond these specialized areas, Whitebox provides a full suite of standard GIS operations, including raster analysis, vector processing, and terrain analysis. However, it's important to understand that Whitebox is designed as an analytical back-end rather than a complete GIS solution. It focuses on data processing and analysis rather than cartographic visualization, which makes it an excellent complement to visualization-focused software like QGIS or ArcGIS.

21.3.3. How is Whitebox different?

Rather than competing with established GIS platforms like QGIS or ArcGIS, Whitebox is designed to extend and enhance their capabilities. You can integrate WhiteboxTools directly into QGIS through plugins or use it alongside ArcGIS to access hundreds of additional analytical tools. This complementary approach means you can leverage the visualization and data management strengths of traditional GIS software while accessing the specialized analytical capabilities of Whitebox.

For Python users, Whitebox Workflows⁸⁴ (WbW) provides a seamless way to incorporate Whitebox functions into scripted workflows. This integration is particularly powerful when combined with other Python geospatial libraries, allowing you to create comprehensive analytical pipelines that handle everything from data acquisition to final visualization.

⁸²<https://jblindsay.github.io/ghrg/index.html>

⁸³<https://www.rust-lang.org>

⁸⁴<https://www.whiteboxgeo.com/whitebox-workflows-for-python>

The performance characteristics of Whitebox also set it apart from many alternatives. The platform is optimized for speed and can handle large datasets efficiently, making it suitable for regional or even continental-scale analyses. This performance advantage becomes particularly apparent when working with high-resolution elevation data or large LiDAR point clouds.

21.4. Useful Resources for Whitebox

As you begin working with WhiteboxTools, several resources will prove invaluable for deepening your understanding and troubleshooting issues:

- [GitHub Repository](#)⁸⁵ - Primary source for the latest code, bug reports, and feature requests
- [WhiteboxGeo](#)⁸⁶ - Main website with comprehensive information about platform capabilities and commercial offerings
- [User Manual](#)⁸⁷ - Detailed technical documentation with thorough explanations of each tool and its parameters
- [Whitebox Workflows for Python](#)⁸⁸ - Seamless integration of Whitebox functions into Python workflows
- [Whitebox Workflows for Python Pro](#)⁸⁹ - Professional version with additional capabilities
- [Python Frontend](#)⁹⁰ - Foundation for Python integration used in this chapter
- [Jupyter Frontend](#)⁹¹ - Interactive interface for exploring tools and parameters
- [R Frontend](#)⁹² - Integration with R programming environment
- [ArcGIS Toolbox](#)⁹³ - Integration with ArcGIS software
- [QGIS Plugin](#)⁹⁴ - Plugin for QGIS integration

21.5. Installing Whitebox

Before we can begin our analysis, we need to set up the necessary software environment. The installation process involves several Python packages that work together to provide both analytical capabilities and interactive visualization features.

The core packages we need include `leafmap`, which provides interactive mapping capabilities and serves as our interface to WhiteboxTools, and `whitebox`, which contains the actual analytical functions. We also need some supporting packages for handling different types of geospatial data, including raster and LiDAR formats.

```
# %pip install whitebox "pygis[lidar]"
```

Once the packages are installed, we can import the necessary libraries and configure our environment for the analyses that follow.

⁸⁵<https://github.com/jblindsay/whitebox-tools>

⁸⁶<https://www.whiteboxgeo.com>

⁸⁷https://whiteboxgeo.com/manual/wbt_book/preface.html

⁸⁸<https://www.whiteboxgeo.com/whitebox-workflows-for-python>

⁸⁹<https://www.whiteboxgeo.com/whitebox-workflows-professional>

⁹⁰<https://github.com/opengeos/whitebox-python>

⁹¹<https://github.com/opengeos/whiteboxgui>

⁹²<https://github.com/opengeos/whiteboxR>

⁹³<https://github.com/opengeos/WhiteboxTools-ArcGIS>

⁹⁴https://plugins.qgis.org/plugins/whitebox_workflows_for_qgis

```
import os
import leafmap
import numpy as np
```

WhiteboxTools generates raster datasets with specific data types and nodata values that need to be handled properly for visualization. Many of the raster outputs use 32-bit integer format with a nodata value of -32768 . By setting this as an environment variable, we ensure that leafmap will correctly interpret and display these datasets.

```
os.environ["NODATA"] = "-32768"
```

21.6. Watershed Analysis

Watershed analysis represents one of the most fundamental applications in environmental geospatial analysis. A watershed, also known as a drainage basin or catchment, is the area of land where all surface water ultimately drains to a common outlet. Understanding watershed boundaries and the flow of water within them is crucial for water resource management, flood prediction, environmental protection, and land use planning.

The process of watershed analysis typically begins with a Digital Elevation Model (DEM), which represents the three-dimensional shape of the Earth's surface. From this elevation data, we can model how water would flow across the landscape under the influence of gravity, identify the paths that water would take, and determine the boundaries that separate different drainage areas.

21.6.1. Create Interactive Maps

Our first step in watershed analysis involves creating an interactive map that will serve as our workspace for visualizing data and results. Interactive maps are essential tools in geospatial analysis because they allow us to explore our data, understand spatial relationships, and communicate findings effectively.

```
m = leafmap.Map()
m.add_basemap("OpenTopoMap")
m.add_basemap("USGS 3DEP Elevation")
m.add_basemap("USGS Hydrography")
m
```

The basemaps we've added serve different purposes in our analysis. OpenTopoMap provides a general topographic context, showing terrain features and elevation patterns. The USGS 3DEP Elevation basemap offers a detailed view of elevation data, which will help us understand the terrain we're analyzing. The USGS Hydrography basemap shows existing stream networks and water bodies, which we can later compare with our analytical results to validate our watershed delineation.

21.6.2. Download Watershed Data

For this analysis, we'll focus on the Calapooia River basin in Oregon, which provides an excellent example of a well-defined watershed with clear topographic boundaries. The Calapooia River is a tributary of the

Willamette River and drains a portion of the Oregon Cascade Range, creating distinct elevation gradients that make watershed boundaries relatively easy to identify.

We begin by specifying a point within our area of interest. This point will serve as a reference for downloading watershed boundary data from existing databases.

```
lat = 44.361169
lon = -122.821802

m = leafmap.Map(center=[lat, lon], zoom=10)
m.add_marker([lat, lon])
m
```

The coordinates we've chosen represent a location within the Calapooia River basin. By centering our map on this point and adding a marker, we can visualize exactly where our analysis will focus. The zoom level of 10 provides a good balance between showing the local area detail and maintaining context of the broader region.

Now we'll use this point to download existing watershed boundary data. The `leafmap.get_wbd()` function accesses the USGS Watershed Boundary Dataset, which provides standardized watershed delineations at various scales. The `digit=10` parameter specifies that we want HUC-10 level watersheds, which typically represent watersheds of 40,000 to 250,000 acres.

```
geometry = {"x": lon, "y": lat}
gdf = leafmap.get_wbd(geometry, digit=10, return_geometry=True)
gdf.explore()
```

The watershed boundary data we've downloaded provides us with an official delineation that we can use for comparison with our own analysis. This comparison is important for validating our methods and understanding any differences that might arise from using different elevation data sources or analytical approaches.

```
gdf.to_file("basin.geojson")
```

Saving the watershed boundary as a GeoJSON file ensures that we can easily reload and reuse this data throughout our analysis. GeoJSON is a widely supported format that maintains spatial reference information and can be easily shared or used in other applications.

21.6.3. Download and Display DEM

The foundation of any watershed analysis is high-quality elevation data. For this analysis, we'll use data from the USGS 3D Elevation Program (3DEP), which provides some of the most accurate and up-to-date elevation data available for the United States. The 3DEP program uses advanced LiDAR and photogrammetric techniques to create detailed elevation models with typical vertical accuracies of 10 centimeters or better in non-vegetated areas.

```
array = leafmap.get_3dep_dem(
    gdf,
    resolution=30,
    output="dem.tif",
    dst_crs="EPSG:3857",
    to_cog=True,
    overwrite=True,
)
array
```

The parameters we've specified for downloading the DEM are important for our analysis. The 30-meter resolution provides a good balance between detail and computational efficiency for watershed-scale analysis. Higher resolutions would provide more detail but would also require significantly more processing time and storage space. The coordinate reference system EPSG:3857 (Web Mercator) is commonly used for web mapping applications and ensures compatibility with our interactive maps.

The `to_cog=True` parameter creates a Cloud Optimized GeoTIFF, which is a format optimized for efficient access and visualization. This format includes internal tiling and overviews that make it faster to display at different zoom levels.

```
m.add_raster("dem.tif", palette="terrain", nodata=np.nan, layer_name="DEM")
m
```

Visualizing the DEM on our interactive map allows us to immediately see the topographic characteristics of our study area. The terrain color palette provides an intuitive representation where lower elevations appear in greens and blues, while higher elevations are shown in browns and whites. This visualization helps us understand the overall topographic setting and identify major ridgelines that likely form watershed boundaries.

21.6.4. Get DEM metadata

Understanding the characteristics of our elevation data is crucial for interpreting our analysis results. The metadata provides essential information about the spatial properties, data quality, and technical specifications of our DEM.

```
metadata = leafmap.image_metadata("dem.tif")
metadata
```

The metadata reveals important details such as the exact spatial extent of our data, the coordinate reference system, the data type and bit depth, and any compression settings used. This information helps us understand the precision and limitations of our elevation data, which in turn affects the accuracy of our watershed analysis.

Understanding the elevation range in our study area is also important for setting appropriate parameters in our analysis tools and for interpreting our results in the context of the local topography.

21.6.5. Add colorbar

A colorbar provides essential context for interpreting elevation values in our visualization. Without a colorbar, users can see relative elevation differences but cannot determine actual elevation values, which are important for understanding the scale and magnitude of topographic features.

```
leafmap.image_min_max("dem.tif")
```

```
m.add_colormap(cmap="terrain", vmin="60", vmax=1500, label="Elevation (m)")  
m
```

The elevation range from 60 to 1500 meters represents the topographic diversity within our watershed, from the valley floor where the river exits the basin to the ridge tops that form the watershed boundaries (Figure 61). This significant elevation range creates the hydraulic gradients that drive water flow and make watershed delineation possible.

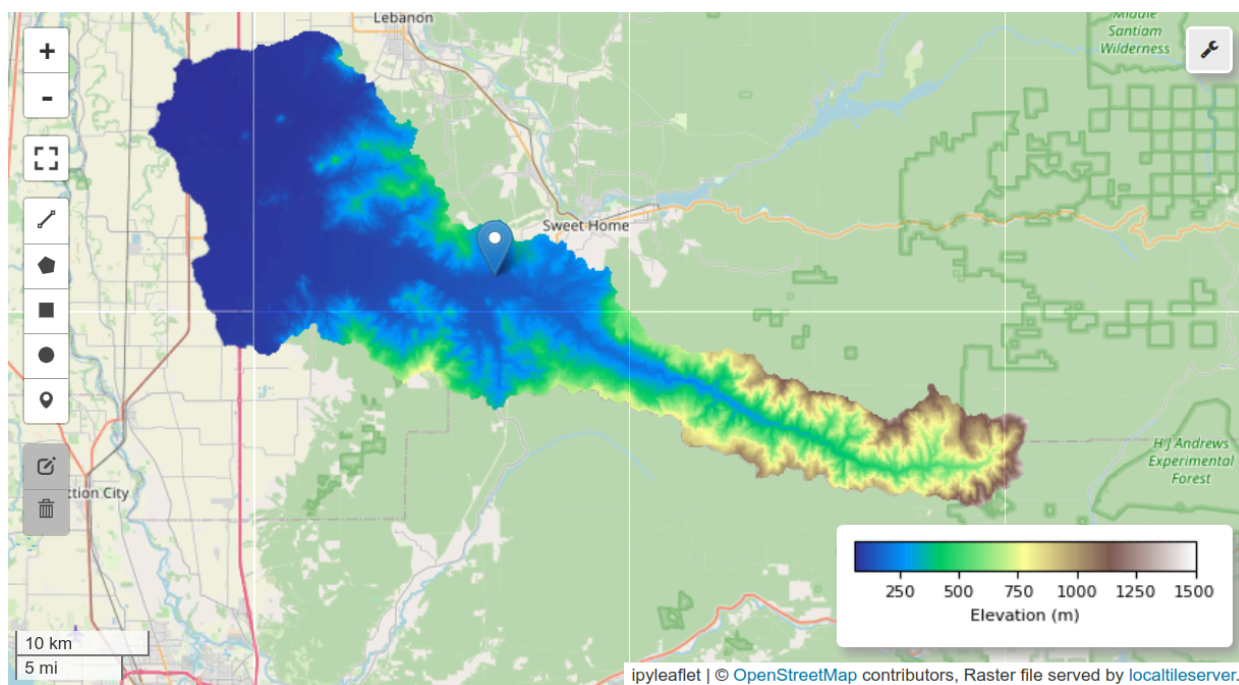


Figure 61: A DEM covering the Calapooia River basin in Oregon.

21.6.6. Initialize WhiteboxTools

Now we can begin using WhiteboxTools for our watershed analysis. The WhiteboxTools class provides our interface to the hundreds of geospatial analysis functions available in the Whitebox platform.

```
wbt = leafmap.WhiteboxTools()
```

Checking the version ensures that we're using a current version of the software and helps with reproducibility of our analysis. Different versions may have different capabilities or may produce slightly different results due to algorithm improvements.

```
wbt.version()
```

The WhiteboxTools interface (Figure 62) provides a comprehensive view of all available tools, organized by category. While we'll use the Python API for our analysis, this interface is valuable for exploring capabilities and understanding tool parameters.

```
leafmap.whiteboxgui()
```

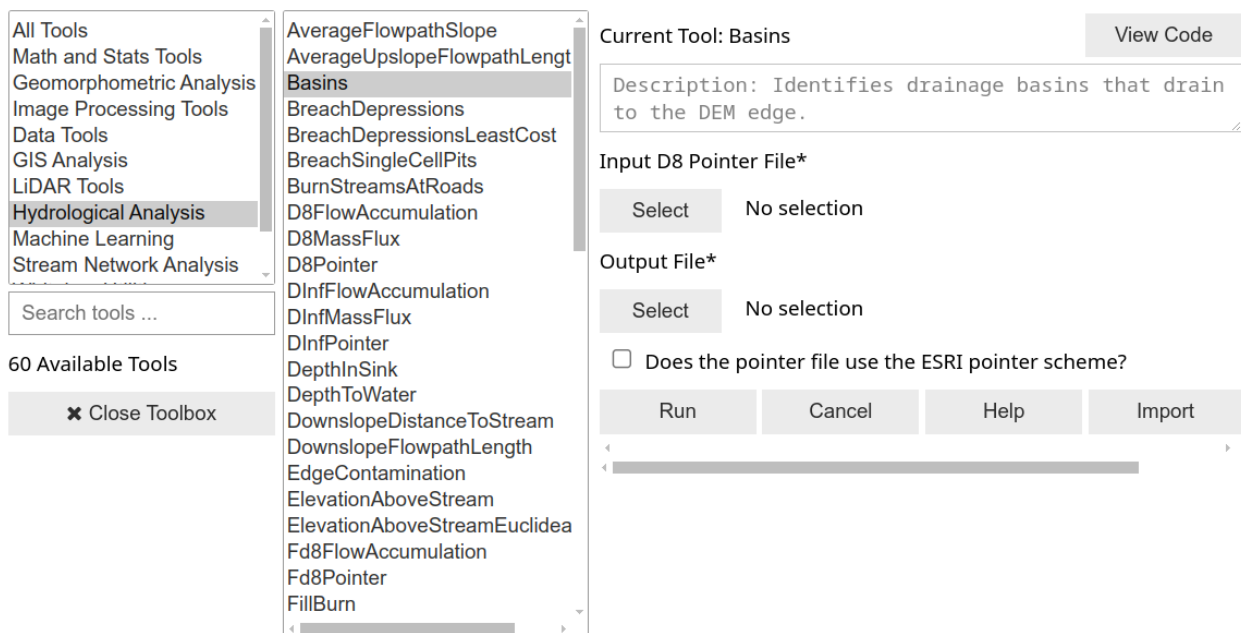


Figure 62: The Whitebox GUI.

21.6.7. Initialize WhiteboxTools

Now we can begin using WhiteboxTools for our watershed analysis. The WhiteboxTools class provides our interface to the hundreds of geospatial analysis functions available in the Whitebox platform.

21.6.8. Set working directory

Organizing our workspace is important for managing the numerous intermediate and output files that watershed analysis typically generates. By setting a working directory, we ensure that all files are created in a predictable location and can be easily managed.

```
wbt.set_working_dir(os.getcwd())
wbt.verbose = False
```


Setting `verbose=False` reduces the amount of processing information displayed, which keeps our notebook output clean and focused on the essential results. However, if you encounter problems with any analysis step, you can set `verbose=True` to get detailed information about what the tools are doing.

21.6.9. Smooth DEM

Real-world elevation data often contains small-scale noise or artifacts that can interfere with hydrological analysis. These might result from measurement errors, vegetation effects, or processing artifacts. Smoothing the DEM can improve the quality of subsequent flow analysis by reducing these irregularities while preserving the major topographic features that control water flow.

```
wbt.feature_preserving_smoothing("dem.tif", "smoothed.tif", filter=9)
```

The feature-preserving smoothing algorithm is specifically designed for elevation data. Unlike simple averaging filters, it attempts to smooth noise while maintaining important topographic features like ridgelines and valley bottoms. The filter size of 9 represents a 9x9 pixel window, which provides moderate smoothing appropriate for 30-meter resolution data.

The return value of 0 indicates that the operation completed successfully. WhiteboxTools functions return 0 for success and 1 for failure, which allows you to check whether operations completed properly in automated workflows.

Let's visualize the smoothed DEM alongside our watershed boundary to see how the smoothing has affected the elevation data.

```
m = leafmap.Map()
m.add_basemap("Satellite")
m.add_raster("smoothed.tif", colormap="terrain", layer_name="Smoothed DEM")
m.add_geojson("basin.geojson", layer_name="Watershed", info_mode=None)
m.add_basemap("USGS Hydrography", show=False)
m
```

The satellite basemap provides a real-world context for our elevation data, allowing us to see how topographic features correspond to visible landscape features like forests, agricultural areas, and urban development. The USGS Hydrography basemap is available but initially hidden, so we can toggle it on later to compare our analytical results with mapped stream networks.

21.6.10. Create hillshade

A hillshade is a visualization technique that simulates the shadows that would be cast by terrain features under specific lighting conditions. This creates a three-dimensional appearance that makes it much easier to interpret topographic features and understand the shape of the landscape.

```
wbt.hillshade("smoothed.tif", "hillshade.tif", azimuth=315, altitude=35)
```

The azimuth parameter of 315 degrees represents lighting from the northwest, which is a common convention in cartography because it creates shadows that most people find intuitive to interpret. The altitude of 35 degrees represents the angle of the light source above the horizon, creating moderate shadows that provide good contrast without being too dramatic.


```
m.add_raster("hillshade.tif", layer_name="Hillshade")
m.layers[-1].opacity = 0.6
```

By overlaying the hillshade with partial transparency (opacity of 0.6), we create a visualization that combines the elevation information from our DEM with the three-dimensional appearance provided by the hillshade. This combination is particularly effective for understanding the topographic context of our watershed analysis.

21.6.11. Find no-flow cells

Before we begin modeling water flow, it's important to identify any cells in our DEM that have undefined flow directions. These “no-flow” cells can occur in flat areas, depressions, or areas where the elevation data has artifacts that prevent the flow direction algorithm from determining where water would flow.

```
wbt.find_no_flow_cells("smoothed.tif", "noflow.tif")
```

The D8 flow direction algorithm, which we'll use later, assigns flow direction based on the steepest downhill path to one of eight neighboring cells. In areas where all neighboring cells have higher or equal elevation, the algorithm cannot determine a flow direction, resulting in no-flow cells.

```
m.add_raster("noflow.tif", layer_name="No Flow Cells")
```

Visualizing the no-flow cells helps us understand where our elevation data might have problems that could affect our watershed analysis. Large areas of no-flow cells might indicate flat areas like lakes or wetlands, or they might indicate data quality issues that need to be addressed.

21.6.12. Fill depressions

Depressions in elevation data can prevent water from flowing to the watershed outlet, creating unrealistic ponding in the flow model. In reality, water would either fill these depressions until they overflow or would find alternative flow paths. The depression filling process modifies the elevation data to ensure that every cell has a path to the watershed outlet.

```
wbt.fill_depressions("smoothed.tif", "filled.tif")
```

Depression filling works by raising the elevation of cells within depressions to the level of the lowest point on the depression's rim. This creates a continuous flow path while making minimal changes to the original elevation data.

An alternative approach is depression breaching, which creates channels through the barriers that create depressions rather than filling the depressions themselves. This approach can be more appropriate in some situations because it preserves the original elevation values in most areas.

```
wbt.breach_depressions("smoothed.tif", "breached.tif")
```

Let's check how depression breaching affects the no-flow cells in our dataset.

```
wbt.find_no_flow_cells("breached.tif", "noflow2.tif")
```

```
m.layers[-1].visible = False  
m.add_raster("noflow2.tif", layer_name="No Flow Cells after Breaching")  
m
```

Comparing the no-flow cells before and after depression breaching shows us how effective the preprocessing has been. Ideally, we should see a significant reduction in no-flow cells, indicating that our elevation data is now suitable for flow modeling.

21.6.13. Delineate flow direction

Flow direction is fundamental to all subsequent hydrological analysis. The D8 algorithm determines the direction of steepest descent from each cell to one of its eight neighbors. This creates a flow direction grid that represents how water would move across the landscape under the influence of gravity.

```
wbt.d8_pointer("breached.tif", "flow_direction.tif")
```

The D8 algorithm is called “D8” because it considers eight possible flow directions (the four cardinal directions plus the four diagonal directions). While more sophisticated algorithms exist that can split flow among multiple directions, D8 remains popular because it’s computationally efficient and produces results that are easy to interpret and use in subsequent analyses.

The flow direction grid uses a coding system where each direction is represented by a specific value. Understanding this coding system isn’t essential for basic watershed analysis, but it becomes important if you need to interpret the flow direction values directly or use them in custom analyses.

21.6.14. Calculate flow accumulation

Flow accumulation represents the number of cells that drain through each cell in the grid. Cells with high flow accumulation values are located in valleys or channels where water from many upslope cells converges. These high-accumulation areas typically correspond to streams and rivers in the real landscape.

```
wbt.d8_flow_accumulation("breached.tif", "flow_accum.tif")
```

The flow accumulation calculation follows the flow directions we calculated in the previous step, counting how many cells contribute flow to each location. Cells at ridge tops have low flow accumulation (only their own area), while cells in major valleys have high flow accumulation (representing the drainage from large upslope areas).

```
m.add_raster("flow_accum.tif", layer_name="Flow Accumulation")
```

Visualizing flow accumulation reveals the drainage network structure within our watershed. The brightest areas represent locations where the most water would accumulate, typically corresponding to the main

river channels and major tributaries. This pattern should generally match the actual stream network visible in satellite imagery or hydrography datasets.

21.6.15. Extract streams

Stream extraction uses the flow accumulation data to identify cells that likely represent actual streams or rivers. The process involves setting a threshold value: cells with flow accumulation above this threshold are classified as streams, while cells below the threshold are classified as hillslopes.

```
wbt.extract_streams("flow_accum.tif", "streams.tif", threshold=5000)
```

The threshold value of 5000 means that a cell must have at least 5000 upslope cells draining through it to be classified as a stream. This threshold determines the density of the extracted stream network. Lower thresholds produce more detailed networks with smaller tributaries, while higher thresholds produce simpler networks with only the major channels.

Selecting an appropriate threshold requires balancing detail with accuracy. Too low a threshold will identify many small channels that may not actually carry water year-round, while too high a threshold may miss important tributaries. The optimal threshold often depends on the resolution of your elevation data, the climate of your study area, and the specific requirements of your analysis.

```
m.layers[-1].visible = False
m.add_raster("streams.tif", layer_name="Streams")
m
```

The extracted stream network should generally match the pattern visible in the flow accumulation data, but with a clear binary classification between stream and non-stream cells. This binary classification is useful for many types of analysis, such as calculating distances to streams or determining stream density within different areas.

21.6.16. Calculate distance to outlet

Understanding how far each location is from the watershed outlet provides important information about travel times, sediment transport, and ecological connectivity within the watershed. The distance to outlet calculation follows the flow paths from each cell to determine the flow distance to the watershed outlet.

```
wbt.distance_to_outlet(
    "flow_direction.tif", streams="streams.tif", output="distance_to_outlet.tif"
)
```

This analysis uses both the flow direction grid and the stream network to calculate distances along the actual flow paths rather than straight-line distances. This is important because water and sediment must follow the topographically determined flow paths, which are often much longer than direct distances.

```
m.add_raster("distance_to_outlet.tif", layer_name="Distance to Outlet")
```

The distance to outlet visualization typically shows a pattern where areas near the watershed outlet have low values (short distances) and areas near the watershed boundaries have high values (long distances).

This pattern reflects the dendritic (tree-like) structure of drainage networks, where tributaries converge toward the main channel.

21.6.17. Vectorize streams

While raster representations of streams are useful for many analyses, vector representations are often more convenient for certain applications, such as calculating stream lengths, creating stream network topology, or overlaying streams with other vector datasets.

```
wbt.raster_streams_to_vector(  
    "streams.tif", d8_pntr="flow_direction.tif", output="streams.shp"  
)
```

The vectorization process uses both the stream raster and the flow direction grid to create connected line segments that represent the stream network. The flow direction information is essential for determining how individual stream segments connect to form a coherent network.

There's a known issue with the `raster_streams_to_vector` tool where the output vector file doesn't include coordinate system information. We can fix this by explicitly setting the coordinate reference system.

```
leafmap.vector_set_crs(source="streams.shp", output="streams.shp",  
    crs="EPSG:3857")
```

```
m.add_shp(  
    "streams.shp",  
    layer_name="Streams Vector",  
    style={"color": "#ff0000", "weight": 3},  
    info_mode=None,  
)  
m
```

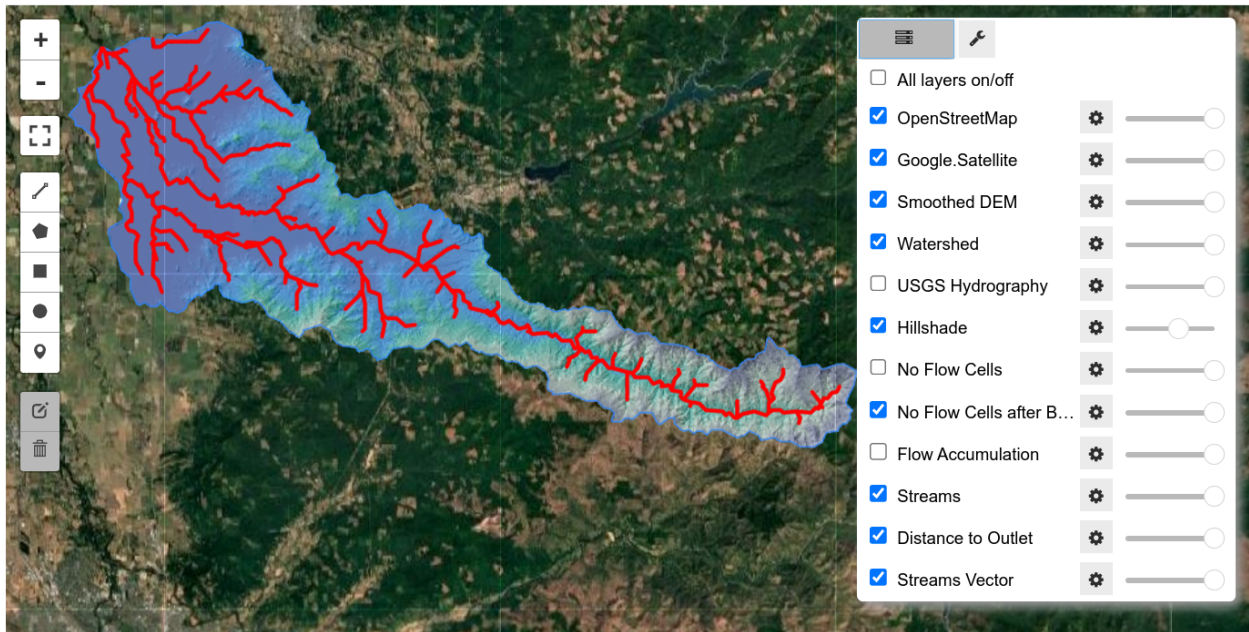


Figure 63: The stream network of the Calapooia River basin.

The vector stream network (Figure 63) provides a clean, cartographic representation that's easy to interpret and compare with other datasets. You can toggle on the USGS Hydrography basemap to compare your extracted stream network with the officially mapped streams, which provides a good validation of your analysis methods and parameters.

21.6.18. Delineate the longest flow path

The longest flow path within a watershed represents the route that water would take to travel the greatest distance from the watershed boundary to the outlet. This path is important for understanding watershed response times and for certain types of hydrological modeling.

```
wbt.basins("flow_direction.tif", "basins.tif")
```

First, we need to identify all the individual drainage basins within our study area. This step creates a raster where each connected drainage area is assigned a unique identifier.

```
wbt.longest_flowpath(
    dem="breached.tif", basins="basins.tif", output="longest_flowpath.shp"
)
```

The longest flow path calculation uses the elevation data and basin boundaries to trace the path from the highest point in each basin to its outlet, following the steepest descent route.

Since we're interested in the single longest flow path for our main watershed, we need to select only the longest path from all the paths that were calculated.

```
leafmap.select_largest(
    "longest_flowpath.shp", column="LENGTH", output="longest_flowpath.shp"
)
```

```
m.add_shp(
    "longest_flowpath.shp",
    layer_name="Longest Flowpath",
    style={"color": "#ff0000", "weight": 3},
)
m
```

The longest flow path provides important information about the watershed's hydrological characteristics. Longer flow paths generally indicate longer travel times for water and sediment, which affects flood timing and water quality processes within the watershed.

21.6.19. Generate a pour point

A pour point is the location where water exits a watershed, and it's essential for watershed delineation. The pour point defines the downstream boundary of the watershed, and all areas that drain to this point will be included in the watershed boundary.

```
if m.user_roi is not None:
    m.save_draw_features("pour_point.shp", crs="EPSG:3857")
else:
    lat = 44.284642
    lon = -122.611217
    leafmap.coords_to_vector([lon, lat], output="pour_point.shp",
    crs="EPSG:3857")
    m.add_marker([lat, lon])
```

You can use the drawing tools in the interactive map to specify your own pour point, or use the default location provided. The choice of pour point significantly affects the resulting watershed boundary, so it's important to select a location that represents a meaningful hydrological boundary, such as a stream gauge location or the confluence with a larger river.

21.6.20. Snap pour point to stream

Pour points should be located precisely on the stream network to ensure accurate watershed delineation. The snapping process moves the pour point to the nearest high-accumulation cell, which should correspond to the actual stream channel.

```
wbt.snap_pour_points(
    "pour_point.shp", "flow_accum.tif", "pour_point_snapped.shp", snap_dist=300
)
```


The snap distance of 300 meters allows the algorithm to search within a reasonable area around the original pour point location. This distance should be large enough to account for small inaccuracies in point placement but not so large that the point gets snapped to an entirely different stream.

```
m.add_shp("pour_point_snapped.shp", layer_name="Pour Point", info_mode=False)
```

Visualizing both the original and snapped pour points helps you verify that the snapping process worked correctly and that the final pour point is located where you intended.

21.6.21. Delineate watershed

Watershed delineation is the culmination of our hydrological analysis. This process uses the flow direction grid and the pour point to identify all cells that drain to the specified outlet location.

```
wbt.watershed("flow_direction.tif", "pour_point_snapped.shp", "watershed.tif")
```

The watershed delineation algorithm works by tracing flow paths backward from the pour point, following the flow direction grid to identify all cells that contribute flow to the outlet. This creates a binary raster where cells within the watershed are assigned one value and cells outside the watershed are assigned another value.

```
m.add_raster("watershed.tif", layer_name="Watershed")  
m
```

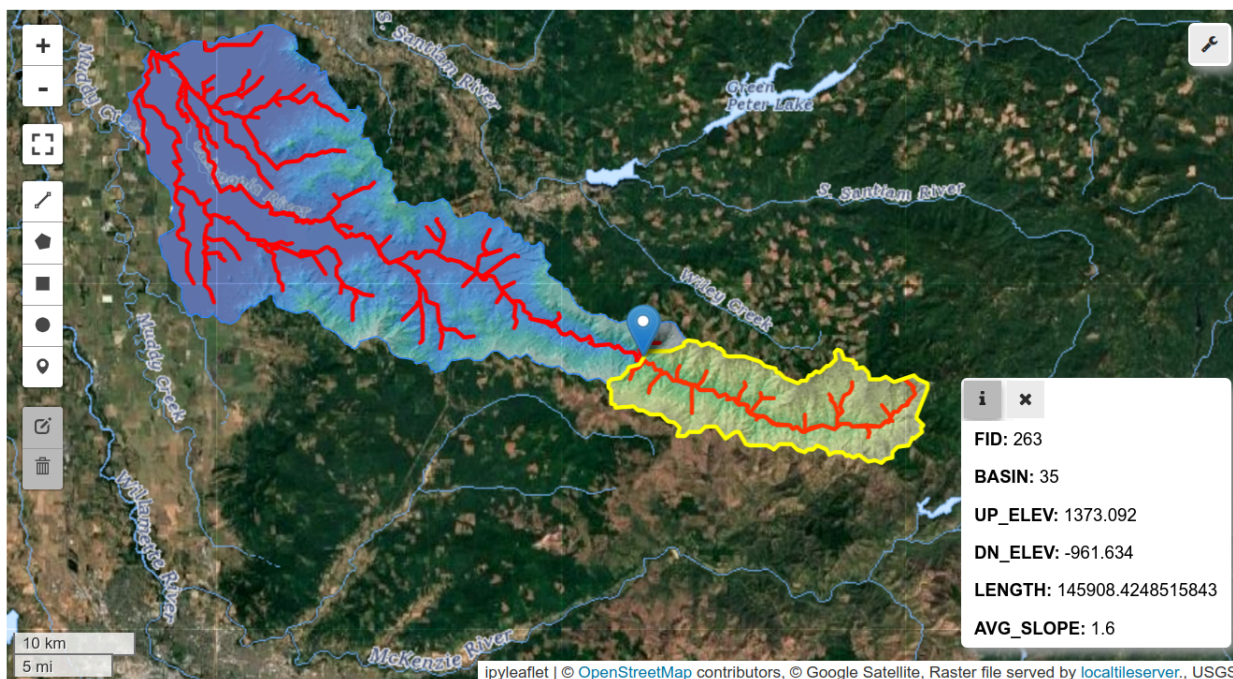


Figure 64: The watershed of the Calapooia River basin.

The delineated watershed boundary (Figure 64) should encompass all the topographic area that drains to your pour point. You can compare this boundary with the original watershed boundary we downloaded earlier to see how well our analysis matches the official delineation.

21.6.22. Convert watershed raster to vector

Converting the watershed boundary from raster to vector format creates a more versatile representation that's easier to use for area calculations, overlay analysis, and cartographic display.

```
wbt.raster_to_vector_polygons("watershed.tif", "watershed.shp")
```

```
m.layers[-1].visible = False
m.add_shp(
    "watershed.shp",
    layer_name="Watershed Vector",
    style={"color": "#ffff00", "weight": 3},
    info_mode=False,
)
```

The vector watershed boundary provides a clean, professional-looking result that clearly delineates the drainage area. This boundary can be used for further analysis, such as calculating watershed area, extracting climate data for the watershed, or analyzing land use patterns within the drainage basin.

21.7. LiDAR Data Analysis

Light Detection and Ranging (LiDAR) technology has revolutionized our ability to measure and analyze three-dimensional landscape features. LiDAR systems use laser pulses to measure distances to the Earth's surface and objects on it, creating detailed point clouds that capture both the ground surface and above-ground features like vegetation and buildings. This section will introduce you to the fundamental concepts and techniques for processing LiDAR data to extract meaningful geospatial information.

LiDAR data analysis typically involves several key steps: data quality assessment and cleaning, classification of points into different categories (ground, vegetation, buildings, etc.), and the creation of various elevation models that represent different aspects of the landscape. The three most common products derived from LiDAR data are Digital Surface Models (DSMs), which represent the top surface including vegetation and buildings; Digital Elevation Models (DEMs), which represent the bare ground surface; and Canopy Height Models (CHMs), which represent the height of vegetation above the ground.

Understanding these different elevation models and how to create them is essential for applications ranging from forest management and urban planning to flood modeling and archaeological research. Each model provides different information about the landscape and is suited to different types of analysis.

21.7.1. Set up whitebox

We begin our LiDAR analysis by setting up the same WhiteboxTools environment we used for watershed analysis. The consistency in setup helps ensure reproducible results and makes it easier to integrate different types of analysis.


```
wbt = leafmap.WhiteboxTools()
wbt.set_working_dir(os.getcwd())
wbt.verbose = False
```

The working directory and verbose settings ensure that our LiDAR processing files are organized in a predictable location and that the output remains clean and focused on essential information.

21.7.2. Download a sample dataset

For this analysis, we'll use a LiDAR dataset from Madison, which provides a good example of mixed urban and natural environments. This dataset includes both ground points and vegetation returns, making it suitable for demonstrating the creation of different types of elevation models.

```
url = "https://github.com/opengeos/datasets/releases/download/lidar/madison.zip"
filename = "madison.las"
leafmap.download_file(url, "madison.zip", quiet=True)
```

LiDAR data is typically distributed in LAS or LAZ format, which are standardized formats specifically designed for storing three-dimensional point cloud data. These formats include not only the x, y, and z coordinates of each point but also additional information such as the intensity of the return signal, the time of measurement, and classification codes that indicate what type of object each point represents.

21.7.3. Read LAS/LAZ data

Loading and inspecting LiDAR data is an important first step that helps us understand the characteristics and quality of our dataset. The inspection process reveals information about the data collection parameters, the spatial extent, the point density, and the types of returns included in the dataset.

```
laz = leafmap.read_lidar(filename)
laz
```

```
<LasData(1.3, point fmt: <PointFormat(1, 0 bytes of extra dims)>, 4068294 points,
2 vlrs)>
```

The LiDAR data object contains both the point data and metadata about the data collection. This metadata includes information about the coordinate system, the data collection date, the sensor specifications, and the data processing history.

```
str(laz.header.version)
```

The LAS format has evolved over time, with different versions supporting different features and capabilities. Understanding the version of your data helps you know what information is available and what processing options are appropriate.

21.7.4. Upgrade file version

Some LiDAR processing tools work better with newer versions of the LAS format. Upgrading to version 1.4 ensures compatibility with the latest processing algorithms and provides access to enhanced features for point classification and analysis.

```
las = leafmap.convert_lidar(laz, file_version="1.4")
str(las.header.version)
```

The version upgrade process maintains all the original point data while updating the file structure to support newer features. This is particularly important for advanced classification and filtering operations.

21.7.5. Write LAS data

Saving the upgraded LAS data ensures that we have a consistent format for all subsequent processing steps. This also creates a backup of our processed data that can be reused if needed.

```
leafmap.write_lidar(las, "madison.las")
```

21.7.6. Histogram analysis

Understanding the distribution of elevation values in your LiDAR data is crucial for setting appropriate parameters in subsequent processing steps. The histogram reveals the range of elevations, the presence of outliers, and the general topographic characteristics of your study area.

```
wbt.lidar_histogram("madison.las", "histogram.html")
```

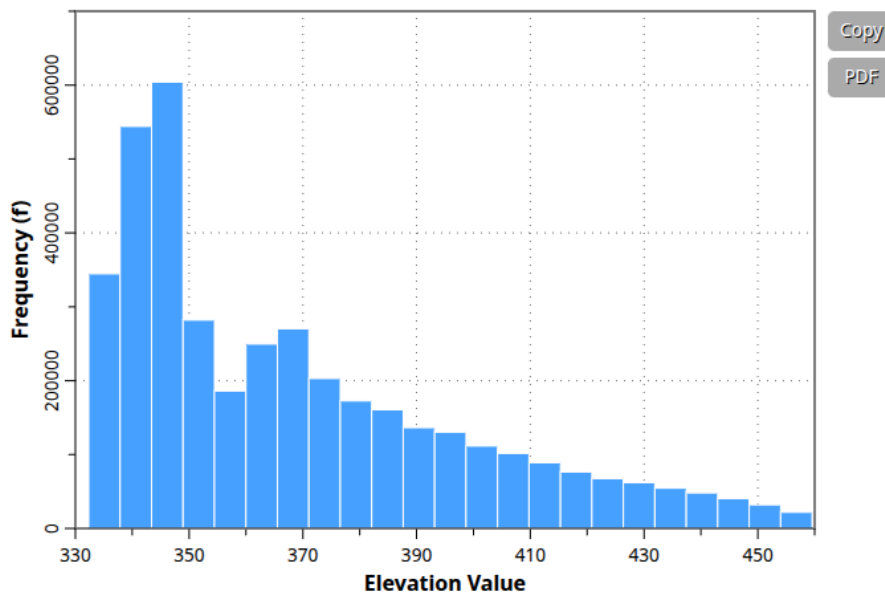


Figure 65: Histogram analysis of the LiDAR dataset.

The histogram analysis creates an interactive HTML file that you can open in a web browser to explore the elevation distribution in detail. This visualization (Figure 65) helps identify potential data quality issues, such as points with unrealistic elevation values that might represent noise or processing errors.

21.7.7. Visualize LiDAR data

Three-dimensional visualization of LiDAR data provides invaluable insights into the structure and characteristics of your dataset. This visualization allows you to see the spatial distribution of points, identify different types of features, and assess the overall quality of the data.

```
leafmap.view_lidar("madison.las")
```

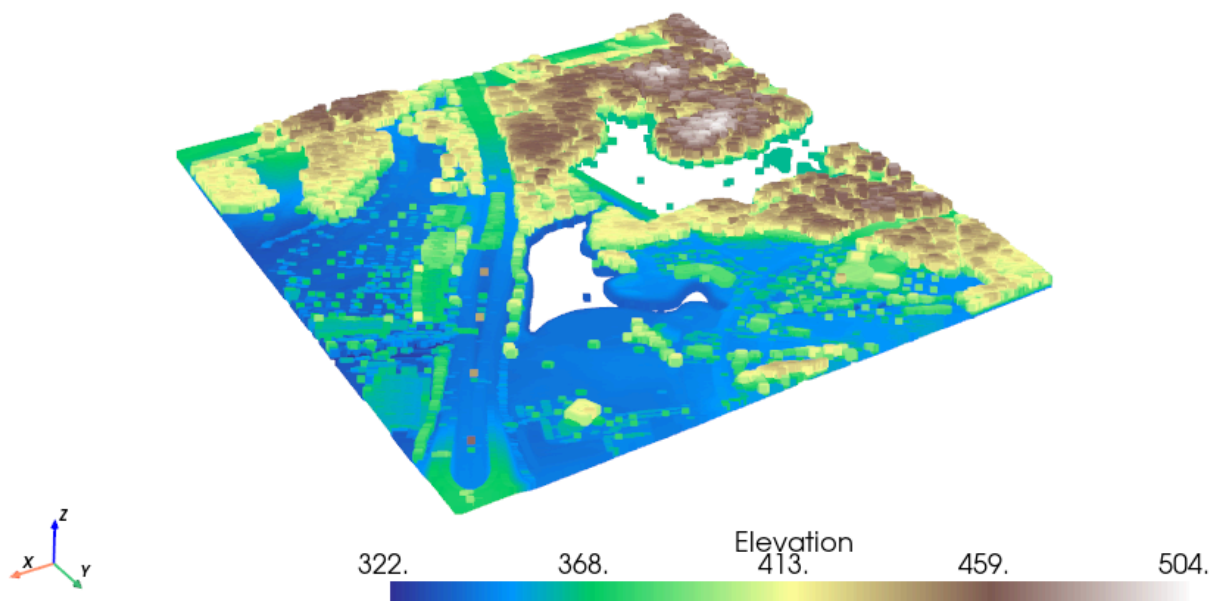


Figure 66: Visualization of the LiDAR dataset.

The 3D visualization (Figure 66) reveals the three-dimensional structure of the landscape, showing both the ground surface and above-ground features like trees and buildings. This perspective is particularly useful for understanding the complexity of the data and for identifying areas that might require special attention during processing.

21.7.8. Remove outliers

LiDAR datasets often contain outlier points that represent measurement errors, atmospheric effects, or reflections from birds or aircraft. These outliers can significantly affect the quality of derived products, so identifying and removing them is an important preprocessing step.

```
wbt.lidar_elevation_slice("madison.las", "madison_rm.las", minz=0, maxz=450)
```

The elevation slice operation removes points outside a specified elevation range. The minimum elevation of 0 meters eliminates points below ground level that likely represent noise, while the maximum elevation of 450 meters removes unrealistically high points that might represent aircraft or atmospheric returns.

Setting appropriate elevation limits requires understanding the topography of your study area. The limits should be wide enough to include all legitimate ground and vegetation points but narrow enough to exclude obvious outliers.

21.7.9. Visualize LiDAR data after removing outliers

Comparing the LiDAR data before and after outlier removal helps verify that the filtering process worked correctly and didn't remove legitimate data points.

```
leafmap.view_lidar("madison_rm.las", cmap="terrain")
```

The terrain colormap provides an intuitive visualization where elevation is represented by color, making it easy to see the topographic structure of the landscape and verify that the outlier removal process preserved the important features while eliminating problematic points.

21.7.10. Create DSM

A Digital Surface Model represents the elevation of the top surface of all features in the landscape, including the ground, vegetation, and buildings. The DSM is created by finding the highest LiDAR return within each grid cell, which typically represents the top of whatever feature is present in that location.

```
wbt.lidar_digital_surface_model(  
    "madison_rm.las", "dsm.tif", resolution=1.0, minz=0, maxz=450  
)
```

The 1-meter resolution provides detailed information while maintaining reasonable file sizes and processing times. Higher resolutions would provide more detail but would also create much larger files and require more processing time. The elevation limits match those used in the outlier removal step, ensuring consistency in our processing workflow.

DSMs are particularly useful for applications that need to consider above-ground features, such as flood modeling in urban areas, viewshed analysis, or solar radiation calculations that must account for buildings and vegetation.

```
leafmap.add_crs("dsm.tif", epsg=2255)
```

WhiteboxTools sometimes creates raster files without coordinate system information, so we need to explicitly set the coordinate reference system to ensure proper spatial referencing for visualization and analysis.

21.7.11. Visualize DSM

Visualizing the DSM allows us to see the three-dimensional structure of the landscape, including both natural and built features.

```
m = leafmap.Map()
m.add_basemap("Satellite")
m.add_raster("dsm.tif", colormap="terrain", layer_name="DSM")
m
```

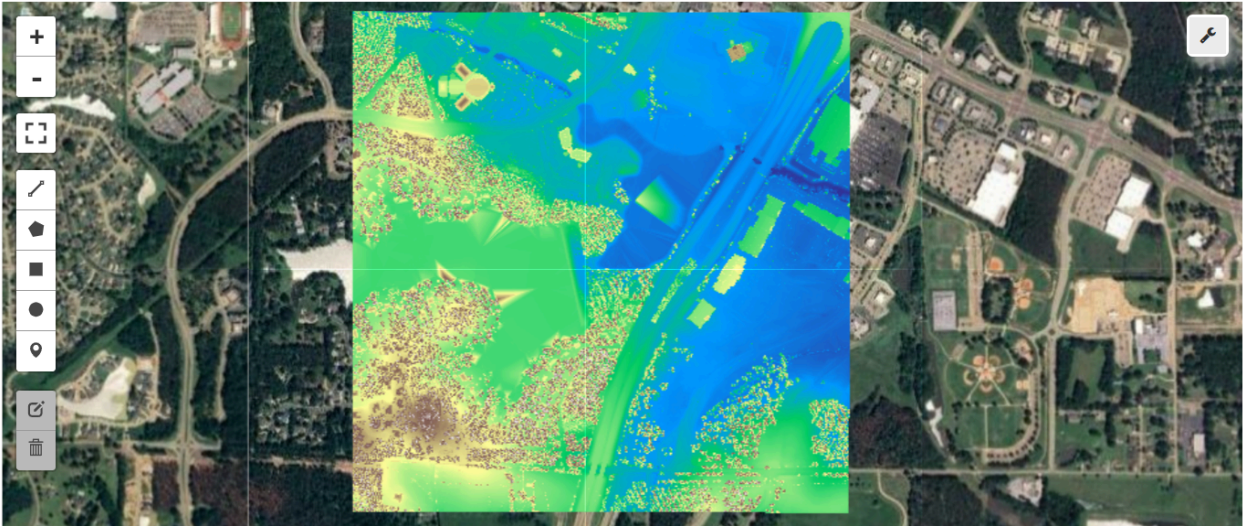


Figure 67: Digital surface model (DSM) of the LiDAR dataset.

The satellite basemap provides important context for interpreting the DSM (Figure 67), allowing you to see how elevation features correspond to visible landscape elements like forests, buildings, and open areas. This comparison helps validate the quality of your DSM and identify any processing artifacts.

21.7.12. Create DEM

A Digital Elevation Model represents the elevation of the bare ground surface, with above-ground features like vegetation and buildings removed. Creating a DEM from a DSM requires algorithms that can distinguish between ground points and above-ground features.

```
wbt.remove_off_terrain_objects("dsm.tif", "dem.tif", filter=25, slope=15.0)
```

The `remove_off_terrain_objects` algorithm identifies and removes features that are likely to be above-ground objects rather than ground surface. The `filter` parameter of 25 specifies the size of the analysis window, while the `slope` parameter of 15.0 degrees helps distinguish between natural terrain variations and artificial features.

This algorithm works by analyzing the local topographic context around each cell. Areas with steep slopes or abrupt elevation changes are more likely to represent above-ground objects and are candidates for removal. The algorithm attempts to interpolate the ground surface beneath these features.

21.7.13. Visualize DEM

The bare-earth DEM provides a representation of the underlying topography without the influence of vegetation or buildings.


```
m.add_raster("dem.tif", palette="terrain", layer_name="DEM")
m
```

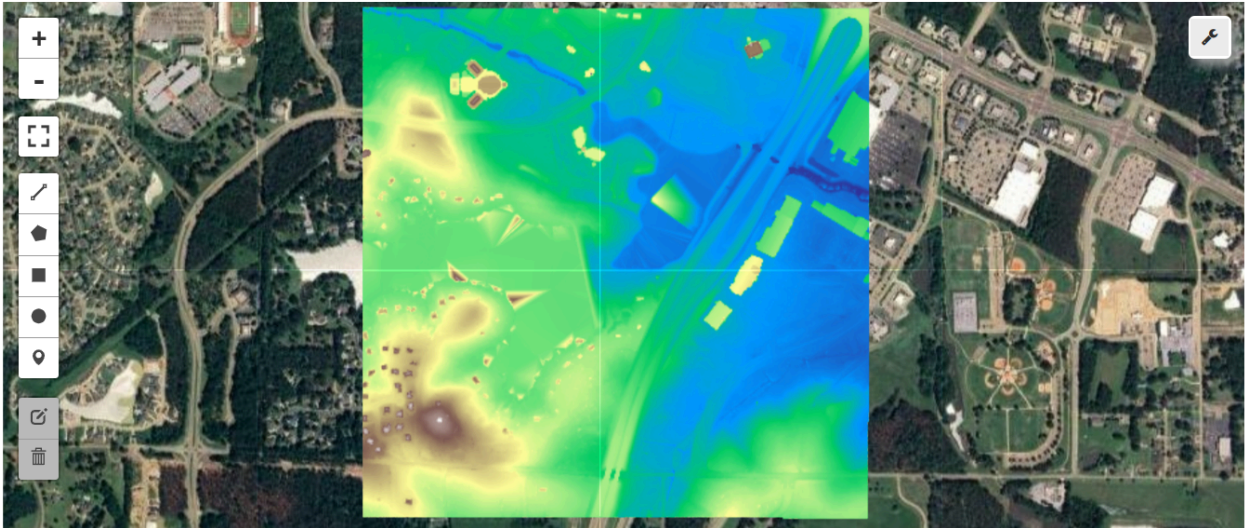


Figure 68: Digital elevation model (DEM) of the LiDAR dataset.

Comparing the DEM (Figure 68) with the DSM reveals the extent and height of above-ground features in your study area. Areas where the two models are similar represent open ground or low vegetation, while areas with large differences represent tall vegetation or buildings.

21.7.14. Create CHM

A Canopy Height Model represents the height of vegetation and other above-ground features by subtracting the ground elevation (DEM) from the surface elevation (DSM). This model is particularly valuable for forest management, ecological studies, and carbon storage assessments.

```
chm = wbt.subtract("dsm.tif", "dem.tif", "chm.tif")
```

The subtraction operation creates a new raster where each cell value represents the height of features above the ground surface. In forested areas, this typically represents tree height, while in urban areas, it might represent building height.

```
m.add_raster("chm.tif", palette="gist_earth", layer_name="CHM")
m.add_layer_manager()
m
```

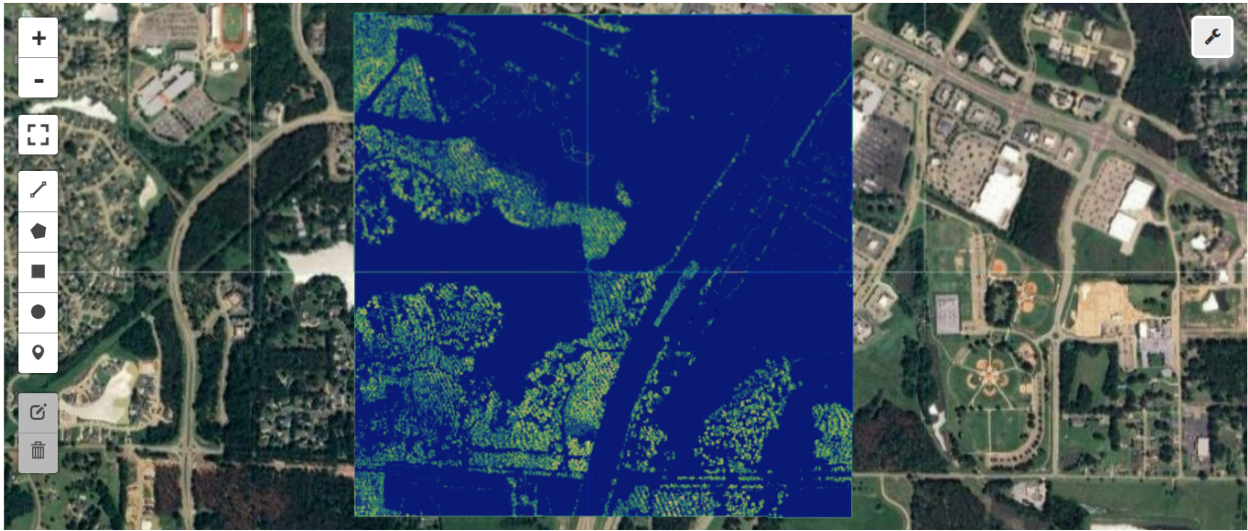


Figure 69: Canopy height model (CHM) of the LiDAR dataset.

The CHM visualization (Figure 69) reveals the three-dimensional structure of vegetation and buildings in your study area. Tall trees and buildings appear as high values, while open areas and low vegetation appear as low values. The layer manager allows you to toggle between different layers to compare the DSM, DEM, and CHM and understand how they relate to each other.

21.8. Key Takeaways

This chapter has provided you with a comprehensive foundation in geospatial analysis using WhiteboxTools, focusing on two fundamental applications that demonstrate the power and versatility of this platform. Through hands-on experience with real-world datasets, you have learned to integrate WhiteboxTools with Python and leafmap to create powerful analytical workflows that combine sophisticated analysis with effective visualization.

The watershed analysis section introduced you to the fundamental concepts of hydrological modeling using Digital Elevation Models. You learned how elevation data can be processed to understand water flow patterns, extract stream networks, and delineate watershed boundaries. These skills are essential for environmental management, water resource planning, and understanding landscape-scale ecological processes. The step-by-step approach you followed, from data preprocessing through final watershed delineation, represents a standard workflow that you can apply to watersheds anywhere in the world.

The LiDAR data analysis section provided you with essential skills for working with three-dimensional point cloud data. You learned to process raw LiDAR data to create Digital Surface Models, Digital Elevation Models, and Canopy Height Models, each of which provides different information about landscape structure. The ability to extract ground surface information and vegetation characteristics from LiDAR data opens up possibilities for applications in forestry, urban planning, archaeology, and many other fields.

The skills you have developed in this chapter provide a solid foundation for more advanced geospatial analysis techniques. Understanding how to work with elevation data, process point clouds, and integrate different types of geospatial information prepares you for tackling complex real-world problems that require sophisticated spatial analysis capabilities.

21.9. Exercises

21.9.1. Exercise 1: Watershed Analysis for a Different Location

In this exercise, you will apply the watershed analysis techniques learned in this chapter to analyze a different watershed. This will help you understand how the methods work across different geographic settings and topographic conditions.

Objective: Perform a complete watershed analysis for the Tualatin River basin in Oregon.

Instructions:

1. **Set up your analysis environment** by importing the necessary libraries and initializing Whitebox-Tools.
2. **Define your study area** using the following coordinates for a point within the Tualatin River basin:
 - Latitude: 45.3311
 - Longitude: -122.7645
3. **Download and prepare elevation data:**
 - Use `leafmap.get_wbd()` to download the watershed boundary (HUC-10 level)
 - Download a 30-meter resolution DEM using `leafmap.get_3dep_dem()`
 - Apply feature-preserving smoothing to reduce noise
4. **Perform hydrological preprocessing:**
 - Create a hillshade for visualization
 - Identify and address no-flow cells using depression breaching
 - Calculate flow direction using the D8 algorithm
5. **Extract hydrological features:**
 - Calculate flow accumulation
 - Extract the stream network using a threshold of 3000 cells
 - Convert the stream network to vector format
 - Calculate the longest flow path
6. **Delineate the watershed:**
 - Create a pour point at coordinates (45.2891, -122.7234)
 - Snap the pour point to the nearest stream
 - Delineate the watershed boundary
 - Convert the watershed to vector format
7. **Create visualizations:**
 - Create an interactive map showing the DEM, hillshade, stream network, and watershed boundary
 - Add appropriate legends and colorbars
 - Compare your results with the official watershed boundary

Questions to consider:

- How does the topography of the Tualatin River basin differ from the Calapooia River basin?
- What threshold value produces a stream network that best matches visible drainage patterns?
- How well does your delineated watershed match the official boundary, and what might cause any differences?

21.9.2. Exercise 2: LiDAR Analysis for Forest Structure Assessment

This exercise focuses on using LiDAR data to assess forest structure and canopy characteristics, which are important for ecological studies and forest management applications.

Objective: Process LiDAR data to create elevation models and analyze vegetation structure in a forested area.

Instructions:

1. Download and inspect LiDAR data:

- Download a different LiDAR dataset from the [USGS 3DEP Program](https://www.usgs.gov/3d-elevation-program/about-3dep-products-services)⁹⁵
- Extract and load the LAS file
- Examine the data characteristics including point density, elevation range, and data version

2. Data quality assessment:

- Create a histogram of elevation values to identify the data distribution
- Visualize the raw point cloud in 3D to understand the landscape structure
- Identify and remove outlier points (use appropriate elevation limits based on your data inspection)

3. Generate elevation models:

- Create a Digital Surface Model (DSM) at 2-meter resolution
- Generate a bare-earth Digital Elevation Model (DEM) using the `remove_off_terrain_objects` tool
- Calculate a Canopy Height Model (CHM) by subtracting the DEM from the DSM

4. Analyze forest structure:

- Calculate summary statistics for the CHM (minimum, maximum, mean, and standard deviation of canopy heights)
- Create a histogram of canopy heights to understand the forest structure
- Identify areas with different canopy height classes:
 - Low vegetation (0-5 meters)
 - Medium vegetation (5-15 meters)
 - Tall vegetation (>15 meters)

5. Create visualizations:

- Display all three elevation models (DSM, DEM, CHM) on an interactive map
- Use appropriate color schemes for each model type
- Add a satellite basemap for context
- Create a layer manager to toggle between different models

6. Advanced analysis (optional):

- Calculate the percentage of area covered by each vegetation height class
- Identify the tallest trees in the dataset and their locations
- Create a slope map from the DEM to understand terrain characteristics

Questions to consider:

- What does the CHM reveal about the forest structure and species composition?
- How do the elevation models differ in areas with dense vegetation versus open areas?

⁹⁵<https://www.usgs.gov/3d-elevation-program/about-3dep-products-services>

- What are the advantages and limitations of using LiDAR-derived DEMs compared to photogrammetric DEMs?
- How might the resolution of your elevation models affect the accuracy of forest structure analysis?

Chapter 22. 3D Mapping with MapLibre

22.1. Introduction

This chapter introduces the [MapLibre](#)⁹⁶ mapping backend in the Leafmap library—a powerful open-source Python tool for creating customizable 2D and 3D interactive maps. With MapLibre, GIS professionals can develop tailored visualizations that enhance data presentation and geospatial analysis. The provided Jupyter notebook demonstrates foundational skills, including interactive map creation, basemap customization, and data layer integration. Additionally, advanced features such as 3D building visualizations and dynamic map controls equip students with practical skills for effective geospatial data visualization and analysis.

22.2. Learning Objectives

By the end of this chapter, students will be able to:

1. **Set up and install MapLibre** for Python-based geospatial visualization.
2. **Create interactive maps** and apply various basemap styles.
3. **Customize map elements**, including markers, lines, polygons, and interactive controls.
4. **Explore advanced features** like 3D building models and choropleth maps.
5. **Integrate and manage multiple data layers**, such as GeoJSON, raster, and vector formats.
6. **Export interactive maps** as standalone HTML files for easy sharing and web deployment.

22.3. Useful Resources

- [MapLibre GL JS Documentation](#)⁹⁷: Comprehensive documentation for MapLibre GL JS.
- [MapLibre Python Bindings](#)⁹⁸: Information on using MapLibre with Python.
- [MapLibre in Leafmap](#)⁹⁹: Examples and tutorials for MapLibre in Leafmap.
- [Video Tutorials](#)¹⁰⁰: Video guides for practical MapLibre skills.
- [MapLibre Demos](#)¹⁰¹: Interactive demos showcasing MapLibre’s capabilities.

22.4. Installation and Setup

To install the required packages, uncomment and run the line below.

```
# %pip install -U leafmap pygis
```

Once installed, import the `maplibregl` backend from the `leafmap` package:

```
import leafmap.maplibregl as leafmap
```

⁹⁶<https://github.com/eodaGmbH/py-maplibregl>

⁹⁷<https://maplibre.org/maplibre-gl-js/docs>

⁹⁸<https://github.com/eoda-dev/py-maplibregl>

⁹⁹<https://leafmap.org/maplibre/overview>

¹⁰⁰<https://bit.ly/maplibre>

¹⁰¹<https://maps.gishub.org>

22.5. Creating Interactive Maps

22.5.1. Basic Map Setup

Let's start by creating a simple interactive map with default settings. This basic setup provides simple map with the `dark-matter` style on which you can add data layers, controls, and other customizations.

```
m = leafmap.Map()  
m
```

22.5.2. Customizing the Map's Center and Zoom Level

You can specify the map's center (latitude and longitude), zoom level, pitch, and bearing for a more focused view. These parameters help direct the user's attention to specific areas.

```
m = leafmap.Map(center=[-100, 40], zoom=3, pitch=0, bearing=0)  
m
```

Hold down the `Ctrl` key. Click and drag to pan the map.

22.5.3. Choosing a Basemap Style

MapLibre supports several pre-defined basemap styles such as `dark-matter`, `positron`, `voyager`, `liberty`, `demotiles`. You can also use custom basemap URLs for unique styling.

```
m = leafmap.Map(style="positron")  
m
```

[OpenFreeMap](https://openfreemap.org)¹⁰² provides a variety of basemap styles that you can use in your interactive maps. These styles include `liberty`, `bright`, and `positron`.

```
m = leafmap.Map(style="liberty", projection="globe")  
m
```

¹⁰²<https://openfreemap.org>

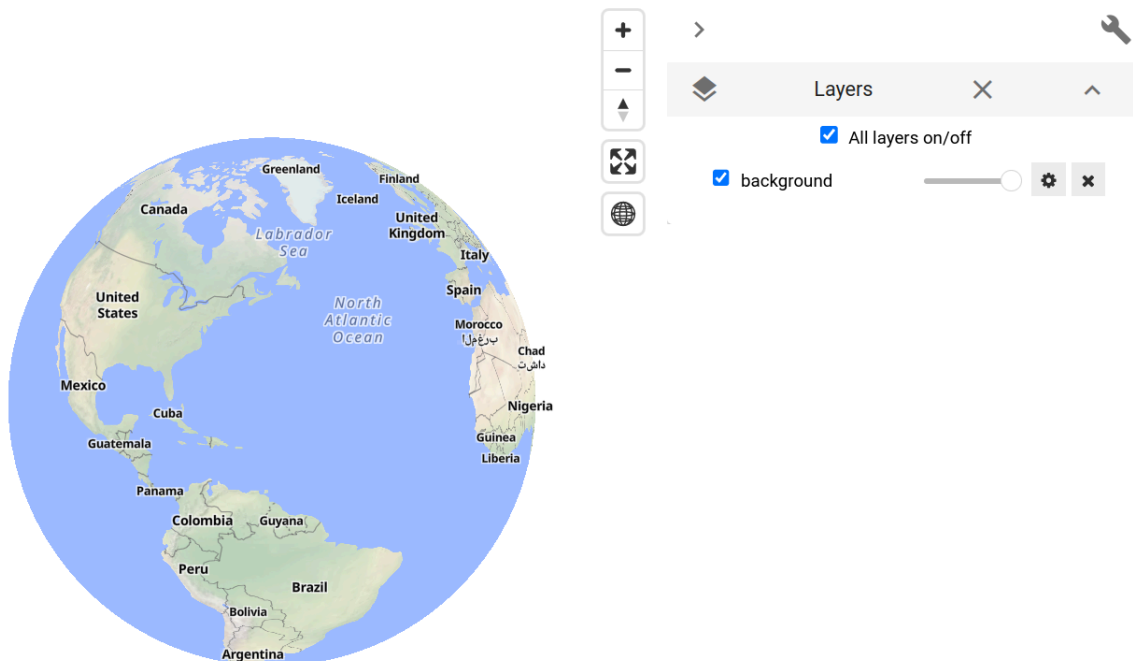


Figure 70: MapLibre with Liberty basemap.

Specifying `projection="globe"` in the map setup creates a 3D globe map (Figure 70). Once the map is displayed, you can click on the globe icon in the upper right corner of the map to toggle the globe view.

Click on the left arrow button in the upper right corner of the map to open the layer manager, which allows you to toggle layer visibility and change the opacity of the layers.

22.5.4. Adding Background Colors

Background color styles, such as `background-lightgray` and `background-green`, are helpful for simple or thematic maps where color-coding is needed.

```
m = leafmap.Map(style="background-lightgray")
m
```

Alternatively, you can provide a URL to a vector style.

```
style = "https://demotiles.maplibre.org/style.json"
m = leafmap.Map(style=style)
m
```

22.6. Adding Map Controls

Map controls enhance the usability of the map by allowing users to interact in various ways, adding elements like scale bars, zoom tools, and drawing options.

22.6.1. Available Controls

- **Geolocate:** Centers the map based on the user's current location, if available.
- **Fullscreen:** Expands the map to a full-screen view for better focus.
- **Navigation:** Provides zoom controls and a compass for reorientation.
- **Draw:** Allows users to draw and edit shapes on the map.

22.6.2. Adding Geolocate Control

The Geolocate control centers the map based on the user's current location, a helpful feature for location-based applications.

```
m = leafmap.Map()
m.add_control("geolocate", position="top-left")
m
```

22.6.3. Adding Fullscreen Control

Fullscreen control enables users to expand the map to full screen, enhancing focus and visual clarity. This is especially useful when viewing complex or large datasets.

```
m = leafmap.Map(center=[11.255, 43.77], zoom=13, style="positron", controls={})
m.add_control("fullscreen", position="top-right")
m
```

22.6.4. Adding Navigation Control

The Navigation control provides buttons for zooming and reorienting the map, improving the user's ability to navigate efficiently.

```
m = leafmap.Map(center=[11.255, 43.77], zoom=13, style="positron", controls={})
m.add_control("navigation", position="top-left")
m
```

22.6.5. Adding Draw Control

The Draw control enables users to interact with the map by adding shapes such as points, lines, and polygons. This control is essential for tasks requiring spatial data input directly on the map.

```
m = leafmap.Map(center=[-100, 40], zoom=3, style="positron")
m.add_draw_control(position="top-left")
m
```

Additionally, you can load a GeoJSON FeatureCollection into the Draw control ([Figure 71](#)), which will allow users to view, edit, or interact with pre-defined geographical features, such as boundaries or points of interest.

```

m = leafmap.Map(center=[-100, 40], zoom=3, style="positron")
geojson = {
    "type": "FeatureCollection",
    "features": [
        {
            "id": "abc",
            "type": "Feature",
            "properties": {},
            "geometry": {
                "coordinates": [
                    [
                        [-119.08, 45.95],
                        [-119.79, 42.08],
                        [-107.28, 41.43],
                        [-108.15, 46.44],
                        [-119.08, 45.95],
                    ]
                ],
                "type": "Polygon",
            },
        },
        {
            "id": "xyz",
            "type": "Feature",
            "properties": {},
            "geometry": {
                "coordinates": [
                    [
                        [-103.87, 38.08],
                        [-108.54, 36.44],
                        [-106.25, 33.00],
                        [-99.91, 31.79],
                        [-96.82, 35.48],
                        [-98.80, 37.77],
                        [-103.87, 38.08],
                    ]
                ],
                "type": "Polygon",
            },
        },
    ],
}
m.add_draw_control(position="top-left", geojson=geojson)
m

```

See [Figure 71](#) for an example of the Draw control.

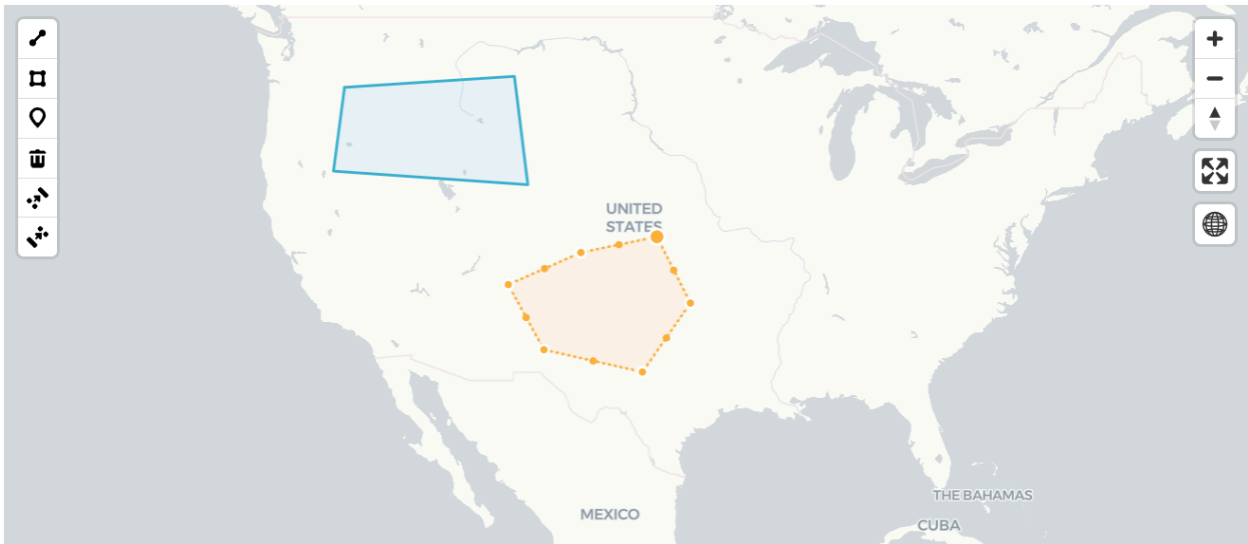


Figure 71: Loading existing GeoJSON data into the map with the Draw control.

Two key methods for accessing drawn features:

- **Selected Features:** Accesses only the currently selected features.
- **All Features:** Accesses all features added, regardless of selection, giving you full control over the spatial data on the map.

```
m.draw_features_selected
```

```
m.draw_feature_collection_all
```

22.7. Adding Layers

Adding layers to a map enhances the data it presents, allowing different types of basemaps, tile layers, and thematic overlays to be combined for in-depth analysis.

22.7.1. Adding Basemaps

Basemaps provide a geographical context for the map. Using the `add_basemap` method, you can select from various basemaps, including `OpenTopoMap` and `Esri.WorldImagery`. Adding a layer control allows users to switch between multiple basemaps interactively.

```
m = leafmap.Map()
m.add_basemap("OpenTopoMap")
m.add_layer_control()
m
```

```
m.add_basemap("Esri.WorldImagery")
```


You can also add basemaps interactively, which provides flexibility for selecting the best background for your map content.

```
m = leafmap.Map()
m
```

```
m.add_basemap()
```

22.7.2. Adding XYZ Tile Layer

XYZ tile layers allow integration of specific tile services like topographic, satellite, or other thematic imagery from XYZ tile servers. By specifying the URL and parameters such as `opacity` and `visibility`, XYZ layers can be customized and styled to fit the needs of your map.

```
m = leafmap.Map()
url = "https://basemap.nationalmap.gov/arcgis/rest/services/USGSTopo/MapServer/
tile/{z}/{y}/{x}"
m.add_tile_layer(url, name="USGS TOp", attribution="USGS", opacity=1.0,
visible=True)
m
```

22.7.3. Adding WMS Layer

Web Map Service (WMS) layers provide access to external datasets served by web servers, such as thematic maps or detailed satellite imagery. Adding a WMS layer involves specifying the WMS URL and layer names, which allows for overlaying various data types such as natural imagery or land cover classifications.

```
m = leafmap.Map(center=[-74.5447, 40.6892], zoom=8, style="liberty")
url = "https://img.nj.gov/imagerywms/Natural2015"
layers = "Natural2015"
m.add_wms_layer(url, layers=layers, before_id="aeroway_fill")
m.add_layer_control()
m
```

```
m = leafmap.Map(center=[-100.307965, 46.98692], zoom=13, pitch=45,
style="liberty")
m.add_basemap("Esri.WorldImagery")
url = "https://fwspublicservices.wim.usgs.gov/wetlandsmapservice/services/
Wetlands/MapServer/WMServer"
m.add_wms_layer(url, layers="1", name="NWI", opacity=0.6)
m.add_layer_control()
m.add_legend(builtin_legend="NWI", title="Wetland Type")
m
```

22.8. Using MapTiler

To use MapTiler with this notebook, you need to set up a MapTiler API key. You can obtain a free API key by signing up at <https://cloud.maptiler.com/>¹⁰³.

```
# import os
# os.environ["MAPTILER_KEY"] = "YOUR_API_KEY"
```

Set the API key as an environment variable to access MapTiler's resources. Once set up, you can specify any named style from MapTiler by using the style parameter in the map setup (see [Figure 72](#)).

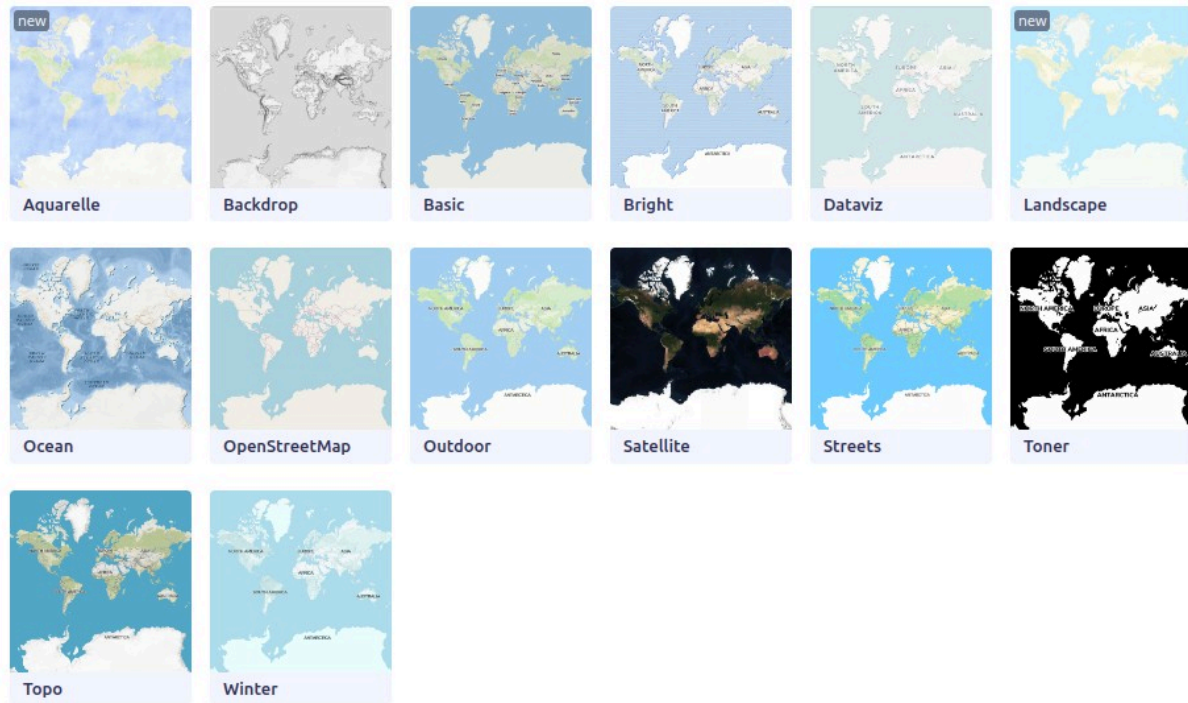


Figure 72: The MapTiler basemaps.

The following are examples of different styles available through MapTiler:

- **Streets style:** This style displays detailed street information and is suited for urban data visualization.
- **Satellite style:** Provides high-resolution satellite imagery for various locations.
- **Hybrid style:** Combines satellite imagery with labels for a clear geographic context.
- **Topo style:** A topographic map showing terrain details, ideal for outdoor-related applications.

```
m = leafmap.Map(style="streets")
m
```

¹⁰³<https://cloud.maptiler.com/>

```
m = leafmap.Map(style="satellite")
m
```

```
m = leafmap.Map(style="hybrid")
m
```

```
m = leafmap.Map(style="topo")
m
```

22.9. 3D Mapping

MapTiler provides a variety of 3D styles that enhance geographic data visualization. Styles can be prefixed with `3d-` to utilize 3D effects such as hybrid, satellite, and topo. The `3d-hillshade` style is used for visualizing hillshade effects alone.

22.9.1. 3D Terrain

These examples demonstrate different ways to implement 3D terrain visualization:

- **3D Hybrid style:** Adds terrain relief to urban data with hybrid visualization.
- **3D Satellite style:** Combines satellite imagery with 3D elevation, enhancing visual context for topography.
- **3D Topo style:** Provides a topographic view with elevation exaggeration and optional hillshade.
- **3D Terrain style:** Displays terrain with default settings for natural geographic areas.
- **3D Ocean style:** A specialized terrain style with oceanic details, using exaggeration and hillshade to emphasize depth.

Each terrain map setup includes a pitch and bearing to adjust the map's angle and orientation, giving a better perspective of 3D features (see [Figure 73](#)).

```
m = leafmap.Map(
    center=[-122.1874314, 46.2022386], zoom=13, pitch=60, bearing=220, style="3d-
    hybrid"
)
m.add_layer_control(bg_layers=True)
m
```



Figure 73: MapTiler 3D hybrid basemap.

```
m = leafmap.Map(
    center=[-122.1874314, 46.2022386],
    zoom=13,
    pitch=60,
    bearing=220,
    style="3d-satellite",
)
m.add_layer_control(bg_layers=True)
m
```

```
m = leafmap.Map(
    center=[-122.1874314, 46.2022386],
    zoom=13,
    pitch=60,
    bearing=220,
    style="3d-topo",
    exaggeration=1.5,
    hillshade=False,
)
m.add_layer_control(bg_layers=True)
m
```

```
m = leafmap.Map(
    center=[-122.1874314, 46.2022386],
    zoom=13,
```

```

    pitch=60,
    bearing=220,
    style="3d-terrain",
)
m.add_layer_control(bg_layers=True)
m

```

```

m = leafmap.Map(style="3d-ocean", exaggeration=1.5, hillshade=True)
m.add_layer_control(bg_layers=True)
m

```

22.9.2. 3D Buildings

Adding 3D buildings enhances urban visualizations, showing buildings with height variations. The setup involves specifying the MapTiler API key for vector tiles and adding building data as a 3D extrusion layer. The extrusion height and color can be set based on data attributes to visualize structures with varying heights, which can be useful in city planning and urban analysis (see [Figure 74](#)).

```

m = leafmap.Map(
    center=[-74.01201, 40.70473], zoom=16, pitch=60, bearing=35, style="basic-v2"
)
MAPTILER_KEY = leafmap.get_api_key("MAPTILER_KEY")
m.add_basemap("Esri.WorldImagery", visible=False)
source = {
    "url": f"https://api.maptiler.com/tiles/v3/tiles.json?key={MAPTILER_KEY}",
    "type": "vector",
}

layer = {
    "id": "3d-buildings",
    "source": "openmaptiles",
    "source-layer": "building",
    "type": "fill-extrusion",
    "min-zoom": 15,
    "paint": {
        "fill-extrusion-color": [
            "interpolate",
            ["linear"],
            ["get", "render_height"],
            0,
            "lightgray",
            200,
            "royalblue",
            400,
            "lightblue",
        ],
    },
},

```

```

    "fill-extrusion-height": [
      "interpolate",
      ["linear"],
      ["zoom"],
      15,
      0,
      16,
      ["get", "render_height"],
    ],
    "fill-extrusion-base": [
      "case",
      [">=", ["get", "zoom"], 16],
      ["get", "render_min_height"],
      0,
    ],
  ],
},
}
m.add_source("openmaptiles", source)
m.add_layer(layer)
m.add_layer_control()
m

```

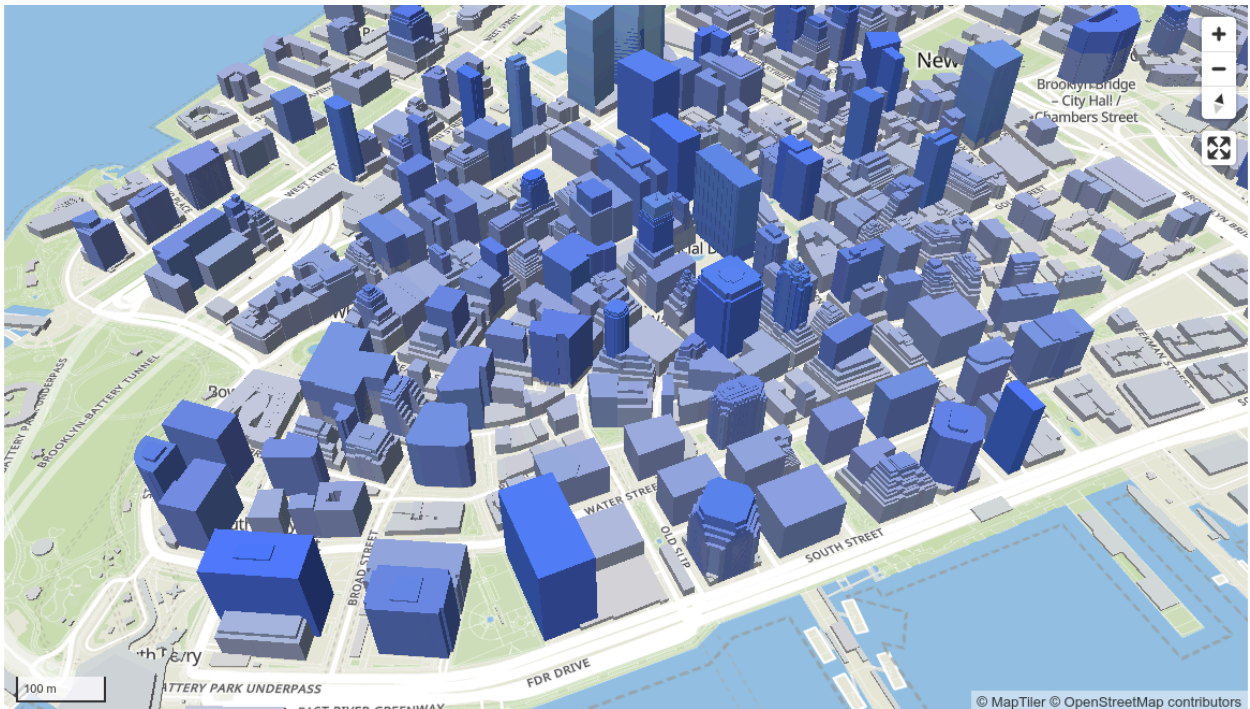


Figure 74: MapTiler 3D buildings basemap.

If you just want to use the MapTiler 3D buildings basemap with the default settings, you can use the following code:


```
m = leafmap.Map(
    center=[-74.01201, 40.70473], zoom=16, pitch=60, bearing=35, style="basic-v2"
)
m.add_basemap("Esri.WorldImagery", visible=False)
m.add_3d_buildings(min_zoom=15)
m.add_layer_control()
m
```

22.9.3. 3D Indoor Mapping

Indoor mapping data can be visualized by loading a GeoJSON file and applying the `add_geojson` method. This setup allows for displaying floorplans with attributes such as color, height, and opacity. It provides a realistic indoor perspective, which is useful for visualizing complex structures or navigating interior spaces.

```
data = "https://maplibre.org/maplibre-gl-js/docs/assets/indoor-3d-map.geojson"
gdf = leafmap.geojson_to_gdf(data)
gdf.explore()
```

```
gdf.head()
```

To visualize indoor spaces in 3D, you can use the `add_geojson` method with the `fill-extrusion` layer type. The following example shows how to visualize a 3D model of a building (Figure 75).

```
m = leafmap.Map(
    center=(-87.61694, 41.86625), zoom=17, pitch=40, bearing=20, style="positron"
)
m.add_basemap("OpenStreetMap.Mapnik")
m.add_geojson(
    data,
    layer_type="fill-extrusion",
    name="floorplan",
    paint={
        "fill-extrusion-color": ["get", "color"],
        "fill-extrusion-height": ["get", "height"],
        "fill-extrusion-base": ["get", "base_height"],
        "fill-extrusion-opacity": 0.5,
    },
)
m.add_layer_control()
m
```

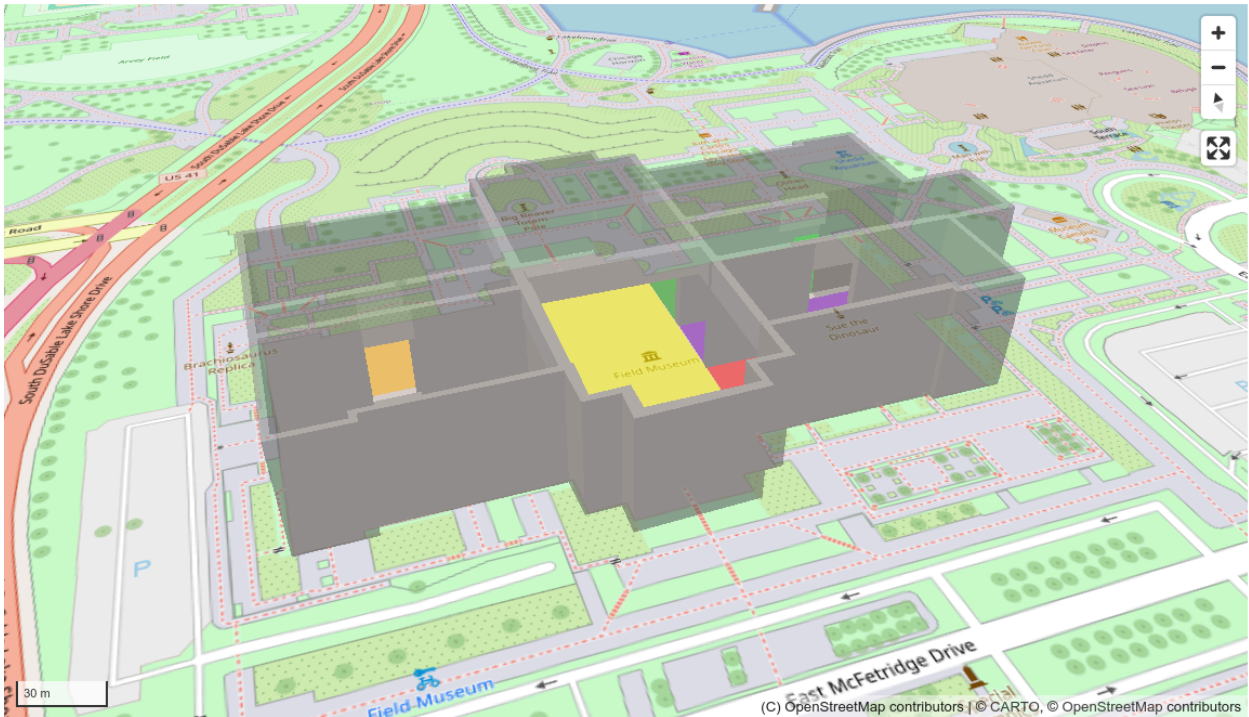


Figure 75: Indoor mapping with MapLibre.

22.9.4. 3D Choropleth Map

Choropleth maps in 3D allow visualizing attribute data by height, where regions are colored and extruded based on specific attributes, such as age or population area. Two examples are provided below:

- **European Countries (Age at First Marriage):** The map displays different colors and extrusion heights based on age data. Color interpolations help represent data variations across regions, making it a suitable map for demographic studies.
- **US Counties (Census Area):** This example uses census data to display county areas in the United States, where each county's area is represented with a different color and height based on its size. This type of map provides insights into geographic distribution and area proportions across the region (see [Figure 76](#)).

Both maps use a fill-extrusion style to dynamically adjust color and height, creating a 3D effect that enhances data interpretation.

```
m = leafmap.Map(center=[19.43, 49.49], zoom=3, pitch=60, style="basic")
source = {
    "type": "geojson",
    "data": "https://docs.maptiler.com/sdk-js/assets/Mean_age_of_women_at_first_
marriage_in_2019.geojson",
}
m.add_source("countries", source)
layer = {
    "id": "eu-countries",
```



```

"source": "countries",
"type": "fill-extrusion",
"paint": {
  "fill-extrusion-color": [
    "interpolate",
    ["linear"],
    ["get", "age"],
    23.0,
    "#fff5eb",
    24.0,
    "#fee6ce",
    25.0,
    "#fdd0a2",
    26.0,
    "#fdae6b",
    27.0,
    "#fd8d3c",
    28.0,
    "#f16913",
    29.0,
    "#d94801",
    30.0,
    "#8c2d04",
  ],
  "fill-extrusion-opacity": 1,
  "fill-extrusion-height": ["*", ["get", "age"], 5000],
},
}
first_symbol_layer_id = m.find_first_symbol_layer()["id"]
m.add_layer(layer, first_symbol_layer_id)
m.add_layer_control()
m

```

```

m = leafmap.Map(center=[-100, 40], zoom=3, pitch=60, style="basic",
projection="globe")
source = {
  "type": "geojson",
  "data": "https://opengeos.org/data/us/us_counties.geojson",
}
m.add_source("counties", source)
layer = {
  "id": "us-counties",
  "source": "counties",
  "type": "fill-extrusion",
  "paint": {
    "fill-extrusion-color": [
      "interpolate",

```

```

    ["linear"],
    ["get", "CENSUSAREA"],
    400,
    "#fff5eb",
    600,
    "#fee6ce",
    800,
    "#fdd0a2",
    1000,
    "#fdae6b",
  ],
  "fill-extrusion-opacity": 1,
  "fill-extrusion-height": ["*", ["get", "CENSUSAREA"], 50],
},
}
first_symbol_layer_id = m.find_first_symbol_layer()["id"]
m.add_layer(layer, first_symbol_layer_id)
m.add_layer_control()
m

```

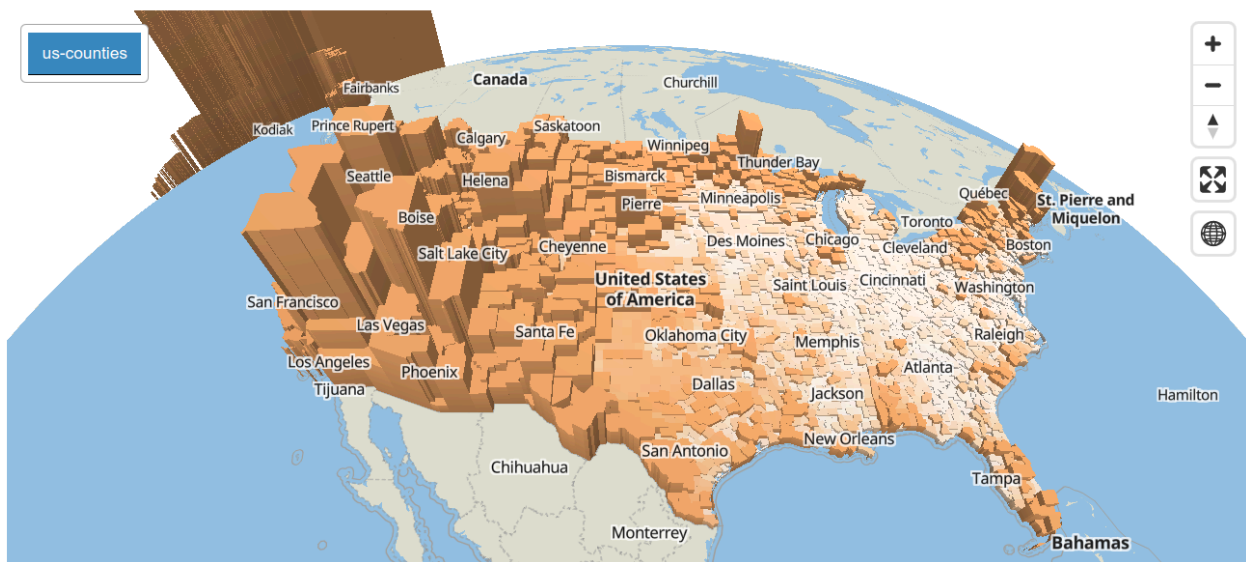


Figure 76: A 3D visualization of US counties.

22.10. Visualizing Vector Data

Leafmap provides a variety of methods to visualize vector data on a map, allowing you to display points, lines, polygons, and other vector shapes with custom styling and interactivity.

22.10.1. Point Data

The following examples demonstrate how to display points on the map.

Simple Marker: Initializes the map centered on a specific latitude and longitude, then adds a static marker to the location.

```
m = leafmap.Map(center=[12.550343, 55.665957], zoom=8, style="positron")
m.add_marker(lng_lat=[12.550343, 55.665957])
m
```

Draggable Marker: Similar to the simple marker, but with an additional option to make the marker draggable. This allows users to reposition it directly on the map.

```
m = leafmap.Map(center=[12.550343, 55.665957], zoom=8, style="positron")
m.add_marker(lng_lat=[12.550343, 55.665957], options={"draggable": True})
m
```

Multiple Points with GeoJSON: Loads point data from a GeoJSON file containing world city locations. After reading the data, it's added to the map as a layer named "cities." Popups can be enabled to display additional information when a user clicks on a city.

```
url = (
    "https://github.com/opengeos/datasets/releases/download/world/world_cities.
    geojson"
)
geojson = leafmap.read_geojson(url)
```

```
m = leafmap.Map(style="streets")
m.add_geojson(geojson, name="cities")
m.add_popup("cities")
m
```

Custom Symbol Layer: Loads point data as symbols instead of markers and customizes the symbol layout using icons, which can be scaled to any size.

```
m = leafmap.Map(style="streets")
source = {"type": "geojson", "data": geojson}

layer = {
    "id": "cities",
    "type": "symbol",
    "source": "point",
    "layout": {
        "icon-image": "marker_15",
        "icon-size": 1,
    },
}
m.add_source("point", source)
m.add_layer(layer)
```

```
m.add_popup("cities")
m
```

22.10.1.1. Customizing Marker Icon Image

To further customize point data, an image icon can be used instead of the default marker:

- **Add Custom Icon:** Loads an OSGeo logo as the custom marker image. The GeoJSON source for conference locations is loaded, and each location is marked with the custom icon and labeled with the year. Layout options define the placement of text and icons.

```
m = leafmap.Map(center=[0, 0], zoom=1, style="positron")
image = "https://maplibre.org/maplibre-gl-js/docs/assets/osgeo-logo.png"
m.add_image("custom-marker", image)

url = "https://github.com/opengeos/datasets/releases/download/places/osgeo_
conferences.geojson"
geojson = leafmap.read_geojson(url)

source = {"type": "geojson", "data": geojson}

m.add_source("conferences", source)
layer = {
    "id": "conferences",
    "type": "symbol",
    "source": "conferences",
    "layout": {
        "icon-image": "custom-marker",
        "text-field": ["get", "year"],
        "text-font": ["Open Sans Semibold", "Arial Unicode MS Bold"],
        "text-offset": [0, 1.25],
        "text-anchor": "top",
    },
}

m.add_layer(layer)
m
```

22.10.2. Line Data

Lines can be displayed to represent routes, boundaries, or connections between locations:

- **Basic Line:** Sets up a line with multiple coordinates to create a path. The map displays this path with rounded line joins and a defined color and width, giving it a polished appearance.

```
m = leafmap.Map(center=[-122.486052, 37.830348], zoom=15, style="streets")
```

```

source = {
  "type": "geojson",
  "data": {
    "type": "Feature",
    "properties": {},
    "geometry": {
      "type": "LineString",
      "coordinates": [
        [-122.48369693756104, 37.83381888486939],
        [-122.48348236083984, 37.83317489144141],
        [-122.48339653015138, 37.83270036637107],
        [-122.48356819152832, 37.832056363179625],
        [-122.48404026031496, 37.83114119107971],
        [-122.48404026031496, 37.83049717427869],
        [-122.48348236083984, 37.829920943955045],
        [-122.48356819152832, 37.82954808664175],
        [-122.48507022857666, 37.82944639795659],
      ],
    },
  },
}

layer = {
  "id": "route",
  "type": "line",
  "source": "route",
  "layout": {"line-join": "round", "line-cap": "round"},
  "paint": {"line-color": "#888", "line-width": 8},
}
m.add_source("route", source)
m.add_layer(layer)
m

```

22.10.3. Polygon Data

Polygons represent regions or areas on the map. Leafmap offers options to display filled polygons with customizable colors and opacities.

- **Basic Polygon:** Adds a GeoJSON polygon representing an area and customizes its fill color and opacity.
- **Outlined Polygon:** In addition to filling the polygon, an outline color is specified to highlight the polygon's boundary. Both fill and line layers are defined, enhancing visual clarity.
- **Building Visualization with GeoJSON:** Uses a URL to load building data and displays it with a customized fill color and outline. The style uses a hybrid map view for additional context.

```

m = leafmap.Map(style="hybrid")
geojson = "https://github.com/opengeos/datasets/releases/download/places/wa_
overture_buildings.geojson"

```

```

paint = {"fill-color": "#ffff00", "fill-opacity": 0.5, "fill-outline-color":
"#ff0000"}
m.add_geojson(geojson, layer_type="fill", paint=paint, name="Fill")
m

```

```

m = leafmap.Map(style="hybrid")
geojson = "https://github.com/opengeos/datasets/releases/download/places/wa_
overture_buildings.geojson"
paint_line = {"line-color": "#ff0000", "line-width": 3}
m.add_geojson(geojson, layer_type="line", paint=paint_line, name="Outline")
paint_fill = {"fill-color": "#ffff00", "fill-opacity": 0.5}
m.add_geojson(geojson, layer_type="fill", paint=paint_fill, name="Fill")
m.add_layer_control()
m

```

22.10.4. Multiple Geometries

This example displays both line and extrusion fills for complex data types:

- **Extruded Blocks with Line Overlay:** Loads Vancouver building blocks data and applies both line and fill-extrusion styles. Each block's height is based on an attribute, which provides a 3D effect. This is useful for visualizing value distribution across an urban landscape (Figure 77).

```

m = leafmap.Map(
    center=[-123.13, 49.254], zoom=11, style="dark-matter", pitch=45, bearing=0
)
url = "https://raw.githubusercontent.com/visgl/deck.gl-data/master/examples/
geojson/vancouver-blocks.json"
paint_line = {
    "line-color": "white",
    "line-width": 2,
}
paint_fill = {
    "fill-extrusion-color": {
        "property": "valuePerSqm",
        "stops": [
            [0, "grey"],
            [1000, "yellow"],
            [5000, "orange"],
            [10000, "darkred"],
            [50000, "lightblue"],
        ],
    },
    "fill-extrusion-height": ["*", 10, ["sqrt", ["get", "valuePerSqm"]]],
    "fill-extrusion-opacity": 0.9,
}

```

```

m.add_geojson(url, layer_type="line", paint=paint_line, name="blocks-line")
m.add_geojson(url, layer_type="fill-extrusion", paint=paint_fill, name="blocks-
fill")
m

```

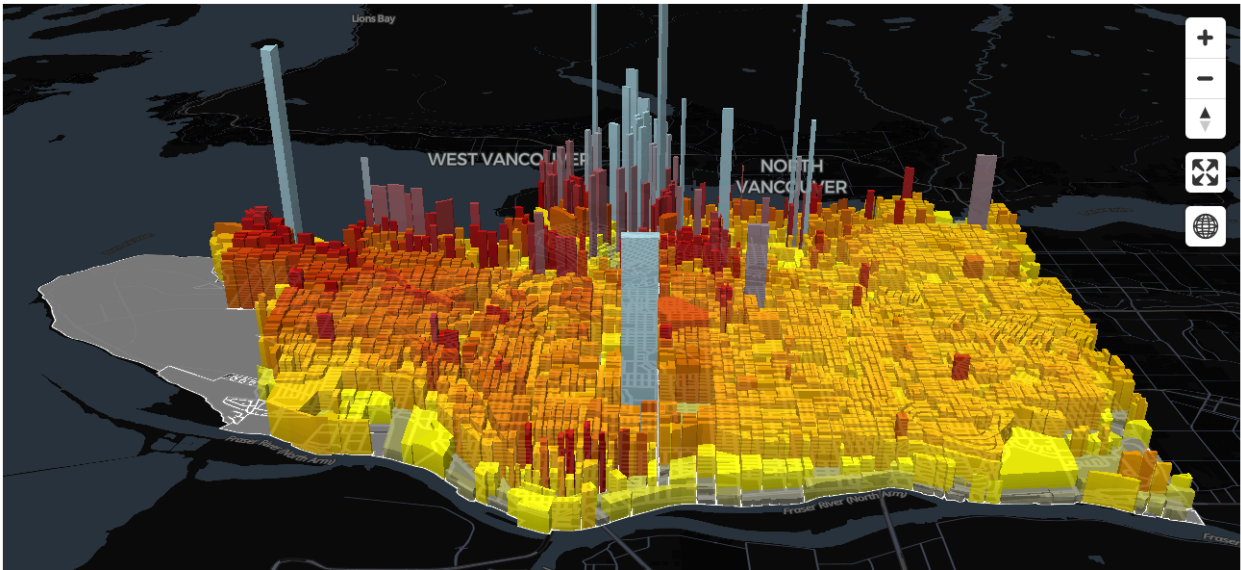


Figure 77: A 3D visualization of building prices in Vancouver, Canada.

22.10.5. Marker Cluster

Clusters help manage large datasets, such as earthquake data, by grouping nearby markers. The color and size of clusters change based on the point count, and filters differentiate individual points from clusters. This example uses nested GeoJSON layers for visualizing clustered earthquake occurrences, with separate styles for individual points and clusters (Figure 78).

```

m = leafmap.Map(center=[-103.59179, 40.66995], zoom=3, style="streets")
data = "https://docs.mapbox.com/mapbox-gl-js/assets/earthquakes.geojson"
source_args = {
    "cluster": True,
    "cluster_radius": 50,
    "cluster_min_points": 2,
    "cluster_max_zoom": 14,
    "cluster_properties": {
        "maxMag": ["max", ["get", "mag"]],
        "minMag": ["min", ["get", "mag"]],
    },
}

m.add_geojson(
    data,

```

```

        layer_type="circle",
        name="earthquake-circles",
        filter=["!", ["has", "point_count"]],
        paint={"circle-color": "darkblue"},
        source_args=source_args,
    )

m.add_geojson(
    data,
    layer_type="circle",
    name="earthquake-clusters",
    filter=["has", "point_count"],
    paint={
        "circle-color": [
            "step",
            ["get", "point_count"],
            "#51bbd6",
            100,
            "#f1f075",
            750,
            "#f28cb1",
        ],
        "circle-radius": ["step", ["get", "point_count"], 20, 100, 30, 750, 40],
    },
    source_args=source_args,
)

m.add_geojson(
    data,
    layer_type="symbol",
    name="earthquake-labels",
    filter=["has", "point_count"],
    layout={
        "text-field": ["get", "point_count_abbreviated"],
        "text-size": 12,
    },
    source_args=source_args,
)
m

```



```

        "building",
        "fill-color",
        ["interpolate", ["exponential", 0.5], ["zoom"], 15, "#e2714b", 22,
"#eee695"],
    )
    m.set_paint_property(
        "building",
        "fill-opacity",
        ["interpolate", ["exponential", 0.5], ["zoom"], 15, 0, 22, 1],
    )
    m.add_layer_control(bg_layers=True)
    m

```

```

m.add_call("zoomTo", 19, {"duration": 9000})

```

22.10.8. Adding a New Layer Below Labels

A layer can be added below existing labels on the map to enhance clarity without obscuring labels. The urban areas dataset is displayed as a fill layer in a color and opacity that visually distinguishes it from other map elements. The layer is positioned below symbols, allowing place names to remain visible.

```

m = leafmap.Map(center=[-88.137343, 35.137451], zoom=4, style="streets")
source = {
    "type": "geojson",
    "data": "https://d2ad6b4ur7yvpq.cloudfront.net/naturalearth-3.3.0/ne_50m_urban_areas.geojson",
}
m.add_source("urban-areas", source)
first_symbol_layer = m.find_first_symbol_layer()
layer = {
    "id": "urban-areas-fill",
    "type": "fill",
    "source": "urban-areas",
    "layout": {},
    "paint": {"fill-color": "#f08", "fill-opacity": 0.4},
}
m.add_layer(layer, before_id=first_symbol_layer["id"])
m

```

22.10.9. Visualizing Population Density

Population density can be calculated and displayed dynamically. This example loads a GeoJSON of Rwandan provinces, calculating density by dividing population by area. The fill color of each province is then adjusted based on density, with different color schemes applied depending on the zoom level.

```

m = leafmap.Map(center=[30.0222, -1.9596], zoom=7, style="streets")
source = {
    "type": "geojson",
    "data": "https://maplibre.org/maplibre-gl-js/docs/assets/rwanda-provinces.
geojson",
}
m.add_source("rwanda-provinces", source)
layer = {
    "id": "rwanda-provinces",
    "type": "fill",
    "source": "rwanda-provinces",
    "layout": {},
    "paint": {
        "fill-color": [
            "let",
            "density",
            ["/", ["get", "population"], ["get", "sq-km"]],
            [
                "interpolate",
                ["linear"],
                ["zoom"],
                8,
                [
                    "interpolate",
                    ["linear"],
                    ["var", "density"],
                    274,
                    ["to-color", "#edf8e9"],
                    1551,
                    ["to-color", "#006d2c"],
                ],
                10,
                [
                    "interpolate",
                    ["linear"],
                    ["var", "density"],
                    274,
                    ["to-color", "#eff3ff"],
                    1551,
                    ["to-color", "#08519c"],
                ],
            ],
        ],
        "fill-opacity": 0.7,
    },
}
m.add_layer(layer)
m

```

22.11. Visualizing Raster Data

22.11.1. Local Raster Data

To visualize local raster files, use the `add_raster` method. In the example, a Landsat image is downloaded and displayed using two different band combinations:

- **Band Combination 3-2-1 (True Color):** Simulates natural colors in the RGB channels.
- **Band Combination 4-3-2:** Enhances vegetation, displaying it in red for better visual contrast. These layers are added to the map along with controls to toggle them. You can adjust brightness and contrast with the `vmin` and `vmax` arguments to improve clarity.

```
url = "https://github.com/opengeos/datasets/releases/download/raster/landsat.tif"
filepath = "landsat.tif"
leafmap.download_file(url, filepath, quiet=True)
```

```
m = leafmap.Map(style="streets")
m.add_raster(filepath, indexes=[3, 2, 1], vmin=0, vmax=100, name="Landsat-321")
m.add_raster(filepath, indexes=[4, 3, 2], vmin=0, vmax=100, name="Landsat-432")
m.add_layer_control()
m
```

A Digital Elevation Model (DEM) is also downloaded and visualized with a terrain color scheme. Leafmap's `layer_interact` method allows interactive adjustments.

```
url = "https://github.com/opengeos/datasets/releases/download/raster/srtm90.tif"
filepath = "srtm90.tif"
leafmap.download_file(url, filepath, quiet=True)
```

```
m = leafmap.Map(style="satellite")
m.add_raster(filepath, colormap="terrain", name="DEM")
m
```

22.11.2. Cloud Optimized GeoTIFF (COG)

Cloud Optimized GeoTIFFs (COG) are large raster files stored on cloud platforms, allowing efficient streaming and loading. This example loads satellite imagery of Libya before and after an event, showing the change over time. Each image is loaded with `add_cog_layer`, and layers can be toggled for comparison. Using `fit_bounds`, the map centers on the COG layer to fit its boundaries.

```
m = leafmap.Map(style="satellite")
before = (
    "https://github.com/opengeos/datasets/releases/download/raster/Libya-2023-07-01.tif"
)
```

```

after = (
    "https://github.com/opengeos/datasets/releases/download/raster/Libya-2023-09-13.tif"
)
m.add_cog_layer(before, name="Before", attribution="Maxar")
m.add_cog_layer(after, name="After", attribution="Maxar", fit_bounds=True)
m.add_layer_control()
m

```

22.11.3. STAC Layer

The SpatioTemporal Asset Catalog (STAC) standard organizes large satellite data collections. With `add_stac_layer`, this example loads Canadian satellite data, displaying both a panchromatic and an RGB layer from the same source. This approach allows easy switching between views.

```

m = leafmap.Map(style="streets")
url = "https://canada-spot-ortho.s3.amazonaws.com/canada_spot_orthoimages/canada_spot5_orthoimages/S5_2007/S5_11055_6057_20070622/S5_11055_6057_20070622.json"
m.add_stac_layer(url, bands=["pan"], name="Panchromatic", vmin=0, vmax=150)
m.add_stac_layer(url, bands=["B4", "B3", "B2"], name="RGB", vmin=0, vmax=150)
m.add_layer_control()
m

```

Leafmap also supports loading STAC items from the [Microsoft Planetary Computer](https://planetarycomputer.microsoft.com)¹⁰⁴. The example demonstrates how to load a STAC item from the Planetary Computer and display it on the map.

```

collection = "landsat-8-c2-l2"
item = "LC08_L2SP_047027_20201204_02_T1"

```

```

leafmap.stac_assets(collection=collection, item=item, titiler_endpoint="pc")

```

```

m = leafmap.Map(style="satellite")
m.add_stac_layer(
    collection=collection,
    item=item,
    assets=["SR_B5", "SR_B4", "SR_B3"],
    name="Color infrared",
)
m

```

¹⁰⁴<https://planetarycomputer.microsoft.com>

22.12. Adding Custom Components

Enhance your maps by adding custom components such as images, videos, text, color bars, and legends.

22.12.1. Adding Image

You can add an image as an overlay or as an icon for a specific layer. For instance:

- Overlaying an image directly on the map at the “bottom-right” corner.
- Adding an icon image to a feature. In the example, a “cat” image is loaded, and a marker is added at coordinates `[0, 0]` with a label “I love kitty!” above the icon.

```
m = leafmap.Map(center=[0.349419, -1.80921], zoom=3, style="streets")
image = "https://upload.wikimedia.org/wikipedia/commons/7/7c/201408_cat.png"
source = {
    "type": "geojson",
    "data": {
        "type": "FeatureCollection",
        "features": [
            {"type": "Feature", "geometry": {"type": "Point", "coordinates": [0,
0]}}
        ],
    },
}

layer = {
    "id": "points",
    "type": "symbol",
    "source": "point",
    "layout": {
        "icon-image": "cat",
        "icon-size": 0.25,
        "text-field": "I love kitty!",
        "text-font": ["Open Sans Regular"],
        "text-offset": [0, 3],
        "text-anchor": "top",
    },
}

m.add_image("cat", image)
m.add_source("point", source)
m.add_layer(layer)
m
```

```
m = leafmap.Map(center=[-100, 40], zoom=3, style="positron")
image = "https://i.imgur.com/LmTETPX.png"
m.add_image(image=image, position="bottom-right")
m
```

```
m = leafmap.Map(center=[-100, 40], zoom=3, style="positron")
content = ''
m.add_html(content, bg_color="transparent", position="bottom-right")
m
```

22.12.2. Adding Text

Add text annotations to the map, specifying parameters like font size and background color. For example:

- Text “Hello World” in the bottom-right corner with a transparent background.
- “Awesome Text!” in the top-left corner with a slightly opaque white background, making it stand out.

```
m = leafmap.Map(center=[-100, 40], zoom=3, style="liberty")
text = "Hello World"
m.add_text(text, fontsize=20, position="bottom-right")
text2 = "Awesome Text!"
m.add_text(text2, fontsize=25, bg_color="rgba(255, 255, 255, 0.8)",
position="top-left")
m
```

22.12.3. Adding GIF

GIFs can be added as animated overlays to bring your map to life. Example: add a sloth GIF in the bottom-right and a second GIF in the bottom-left corner, with a text label indicating “I love sloth!” for added character.

```
m = leafmap.Map(center=[-100, 40], zoom=3, style="positron")
image = "https://i.imgur.com/KeiAsTv.gif"
m.add_image(image=image, width=250, height=250, position="bottom-right")
text = "I love sloth! 🦥 "
m.add_text(text, fontsize=35, padding="20px")
image2 = "https://i.imgur.com/kZC2tpr.gif"
m.add_image(image=image2, bg_color="transparent", position="bottom-left")
m
```

22.12.4. Adding HTML

Embed custom HTML content to display various HTML elements, such as emojis or stylized text. You can also adjust the font size and background transparency for better integration into the map design.

```
m = leafmap.Map(center=[-100, 40], zoom=3, style="positron")
html = """
<html>
<style>
body {
```

```

    font-size: 20px;
}
</style>
<body>

<span style='font-size:100px;'>#128640;</span>
<p>I will display #128641;</p>
<p>I will display #128642;</p>

</body>
</html>
"""
m.add_html(html, bg_color="transparent")
m

```

22.12.5. Adding Color bar

Adding a color bar enhances data interpretation. In the example:

1. A Digital Elevation Model (DEM) is displayed with a color ramp from 0 to 1500 meters.
2. `add_colorbar` method is used to create a color bar with labels, adjusting its position, opacity, and orientation for optimal readability (see [Figure 79](#)).

```
import numpy as np
```

```

m = leafmap.Map(style="topo", height="500px")
dem = "https://github.com/opengeos/datasets/releases/download/raster/dem.tif"
m.add_cog_layer(
    dem,
    name="DEM",
    colormap_name="terrain",
    rescale="0, 1500",
    fit_bounds=True,
    nodata=np.nan,
)
m.add_colorbar(
    cmap="terrain", vmin=0, vmax=1500, label="Elevation (m)", position="bottom-
right"
)
m.add_layer_control()
m

```

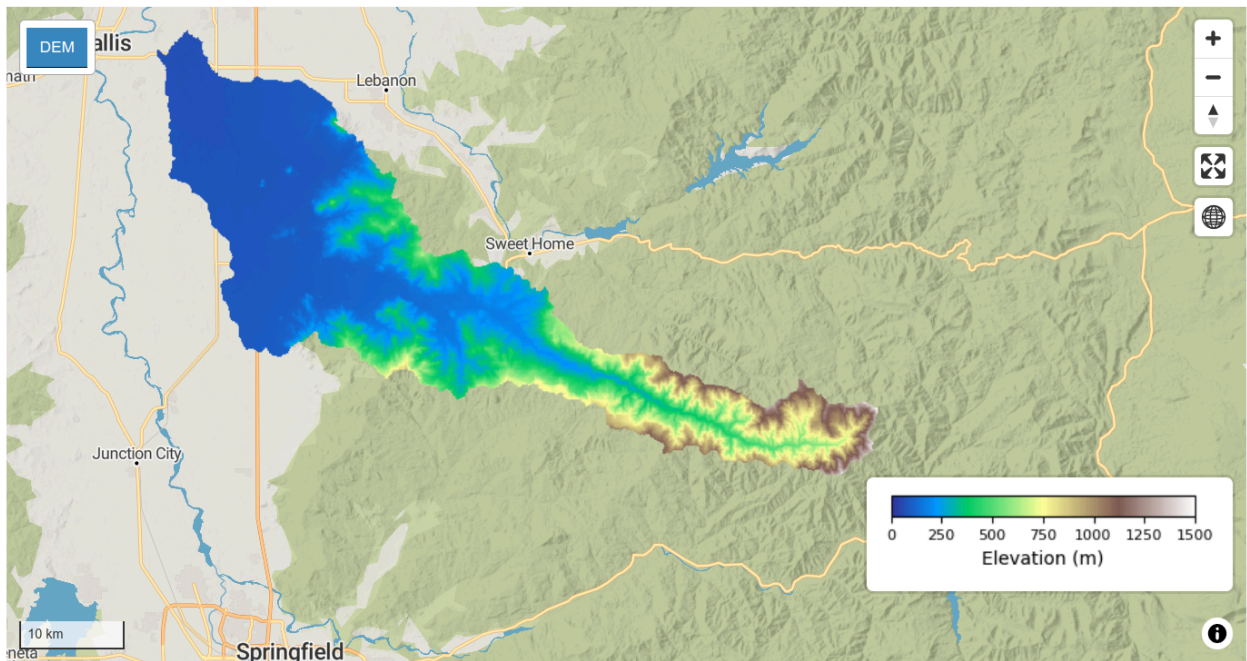



Figure 79: Adding a colorbar to a MapLibre map.

Make the color bar background transparent to blend seamlessly with the map.

```
m = leafmap.Map(style="topo")
m.add_cog_layer(
    dem,
    name="DEM",
    colormap_name="terrain",
    rescale="0, 1500",
    nodata=np.nan,
    fit_bounds=True,
)
m.add_colorbar(
    cmap="terrain",
    vmin=0,
    vmax=1500,
    label="Elevation (m)",
    position="bottom-right",
    transparent=True,
)
m
```

Make the color bar vertical for a different layout.

```
m = leafmap.Map(style="topo")
m.add_cog_layer(
    dem,
```

```

        name="DEM",
        colormap_name="terrain",
        rescale="0, 1500",
        nodata=np.nan,
        fit_bounds=True,
    )
m.add_colorbar(
    cmap="terrain",
    vmin=0,
    vmax=1500,
    label="Elevation (m)",
    position="bottom-right",
    width=0.2,
    height=3,
    orientation="vertical",
)
m

```

22.12.6. Adding Legend

Custom legends help users understand data classifications. Two methods are shown:

1. Using built-in legends, such as for NLCD (National Land Cover Database) or wetland types.
2. Custom legends are built with a dictionary of land cover types and colors. This legend provides descriptive color-coding for various land cover types, with configurable background opacity to blend with the map (see [Figure 80](#)).

```

m = leafmap.Map(center=[-100, 40], zoom=3, style="positron", height="500px")
m.add_basemap("Esri.WorldImagery")
url = "https://www.mrlc.gov/geoserver/mrlc_display/NLCD_2021_Land_Cover_L48/wms"
layers = "NLCD_2021_Land_Cover_L48"
m.add_wms_layer(url, layers=layers, name="NLCD 2021")
m.add_legend(
    title="NLCD Land Cover Type",
    builtin_legend="NLCD",
    bg_color="rgba(255, 255, 255, 0.5)",
    position="bottom-left",
)
m

```

```

m = leafmap.Map(center=[-100, 40], zoom=3, style="positron")
m.add_basemap("Esri.WorldImagery")
url = "https://www.mrlc.gov/geoserver/mrlc_display/NLCD_2021_Land_Cover_L48/wms"
layers = "NLCD_2021_Land_Cover_L48"
m.add_wms_layer(url, layers=layers, name="NLCD 2021")

```

```

legend_dict = {
    "11 Open Water": "466b9f",
    "12 Perennial Ice/Snow": "d1def8",
    "21 Developed, Open Space": "dec5c5",
    "22 Developed, Low Intensity": "d99282",
    "23 Developed, Medium Intensity": "eb0000",
    "24 Developed High Intensity": "ab0000",
    "31 Barren Land (Rock/Sand/Clay)": "b3ac9f",
    "41 Deciduous Forest": "68ab5f",
    "42 Evergreen Forest": "1c5f2c",
    "43 Mixed Forest": "b5c58f",
    "51 Dwarf Scrub": "af963c",
    "52 Shrub/Scrub": "ccb879",
    "71 Grassland/Herbaceous": "dfd9c2",
    "72 Sedge/Herbaceous": "d1d182",
    "73 Lichens": "a3cc51",
    "74 Moss": "82ba9e",
    "81 Pasture/Hay": "dcd939",
    "82 Cultivated Crops": "ab6c28",
    "90 Woody Wetlands": "b8d9eb",
    "95 Emergent Herbaceous Wetlands": "6c9fb8",
}
m.add_legend(
    title="NLCD Land Cover Type",
    legend_dict=legend_dict,
    bg_color="rgba(255, 255, 255, 0.5)",
    position="bottom-left",
)
m

```

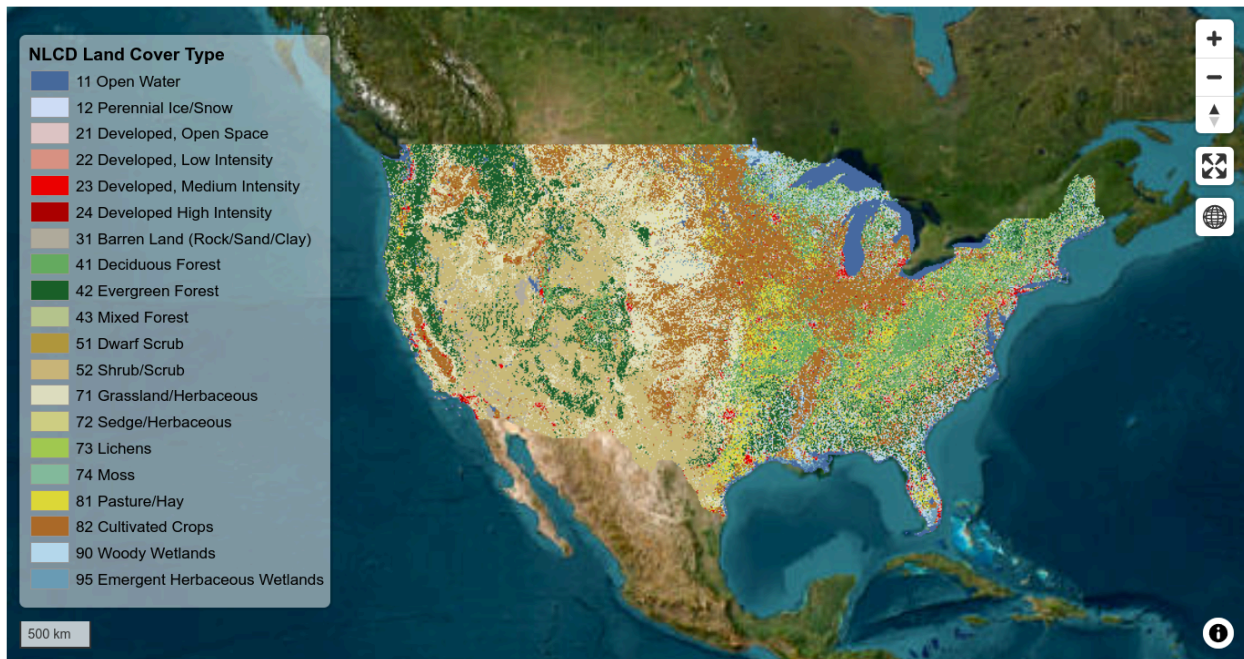


Figure 80: Adding a legend to a MapLibre map.

22.12.7. Adding Video

Videos can be added with geographic context by specifying corner coordinates. Videos must be listed in multiple formats to ensure compatibility across browsers. The coordinates array should define the video's location on the map in the order: top-left, top-right, bottom-right, and bottom-left. This is demonstrated by adding drone footage to a satellite map view, enhancing the user experience with real-world visuals.

```
m = leafmap.Map(
    center=[-122.514426, 37.562984], zoom=17, bearing=-96, style="satellite"
)
urls = [
    "https://static-assets.mapbox.com/mapbox-gl-js/drone.mp4",
    "https://static-assets.mapbox.com/mapbox-gl-js/drone.webm",
]
coordinates = [
    [-122.51596391201019, 37.56238816766053],
    [-122.51467645168304, 37.56410183312965],
    [-122.51309394836426, 37.563391708549425],
    [-122.51423120498657, 37.56161849366671],
]
m.add_video(urls, coordinates)
m.add_layer_control()
m
```

```

m = leafmap.Map(center=[-115, 25], zoom=4, style="satellite")
urls = [
    "https://data.openeos.org/patricia_nasa.mp4",
    "https://data.openeos.org/patricia_nasa.webm",
]
coordinates = [
    [-130, 32],
    [-100, 32],
    [-100, 13],
    [-130, 13],
]
m.add_video(urls, coordinates)
m.add_layer_control()
m

```

22.13. Visualizing PMTiles

Leafmap supports visualizing [PMTiles](https://protomaps.com/docs/pmtiles/)¹⁰⁵, which enables efficient storage and fast rendering of vector tiles directly in the browser.

22.13.1. Building Footprint Data

Visualize the [Google-Microsoft Open Buildings dataset](https://data.source.coop/vida/google-microsoft-open-buildings/pmtiles/go_ms_building_footprints.pmtiles), managed by VIDA, in PMTiles format. Fetch metadata to identify available layers, apply custom styles to the building footprints, and render them with semi-transparent colors for a clear visualization.

```

url = "https://data.source.coop/vida/google-microsoft-open-buildings/pmtiles/go_ms_building_footprints.pmtiles"
metadata = leafmap.pmtiles_metadata(url)
print(f"layer names: {metadata['layer_names']}")
print(f"bounds: {metadata['bounds']}")

```

```

m = leafmap.Map(center=[0, 20], zoom=2)
m.add_basemap("Esri.WorldImagery", visible=False)

style = {
    "version": 8,
    "sources": {
        "example_source": {
            "type": "vector",
            "url": "pmtiles://" + url,
            "attribution": "PMTiles",
        }
    },
}

```

¹⁰⁵<https://protomaps.com/docs/pmtiles/>

```

    "layers": [
      {
        "id": "buildings",
        "source": "example_source",
        "source-layer": "building_footprints",
        "type": "fill",
        "paint": {"fill-color": "#3388ff", "fill-opacity": 0.5},
      },
    ],
  },
}

# style = leafmap.pmtiles_style(url) # Use default style

m.add_pmtiles(
  url,
  style=style,
  visible=True,
  opacity=1.0,
  tooltip=True,
)
m

```

22.13.2. Fields of The World

Visualize the Agricultural Field Boundary dataset - Fields of The World (FTW¹⁰⁶) with MapLibre. The dataset is available on Source Cooperative at <https://source.coop/repositories/kerner-lab/fields-of-the-world/description>¹⁰⁷. The result is shown in Figure 81.

```

url = "https://data.source.coop/kerner-lab/fields-of-the-world/ftw-sources.pmtiles"
metadata = leafmap.pmtiles_metadata(url)
print(f"layer names: {metadata['layer_names']}")
print(f"bounds: {metadata['bounds']}")

```

```

m = leafmap.Map()
# Define colors for each last digit (0-9)
style = {
  "layers": [
    {
      "id": "Field Polygon",
      "source": "example_source",
      "source-layer": "ftw-sources",
      "type": "fill",
    },
  ],
}

```

¹⁰⁶<https://fieldsoftheworld>

¹⁰⁷<https://source.coop/repositories/kerner-lab/fields-of-the-world/description>

```

    "paint": {
      "fill-color": [
        "case",
        ["==", ["%", ["to-number", ["get", "id"]], 10], 0],
        "#FF5733", # Color for last digit 0
        ["==", ["%", ["to-number", ["get", "id"]], 10], 1],
        "#33FF57", # Color for last digit 1
        ["==", ["%", ["to-number", ["get", "id"]], 10], 2],
        "#3357FF", # Color for last digit 2
        ["==", ["%", ["to-number", ["get", "id"]], 10], 3],
        "#FF33A1", # Color for last digit 3
        ["==", ["%", ["to-number", ["get", "id"]], 10], 4],
        "#FF8C33", # Color for last digit 4
        ["==", ["%", ["to-number", ["get", "id"]], 10], 5],
        "#33FFF6", # Color for last digit 5
        ["==", ["%", ["to-number", ["get", "id"]], 10], 6],
        "#A833FF", # Color for last digit 6
        ["==", ["%", ["to-number", ["get", "id"]], 10], 7],
        "#FF333D", # Color for last digit 7
        ["==", ["%", ["to-number", ["get", "id"]], 10], 8],
        "#33FFBD", # Color for last digit 8
        ["==", ["%", ["to-number", ["get", "id"]], 10], 9],
        "#FF9933", # Color for last digit 9
        "#FF0000", # Fallback color if no match
      ],
      "fill-opacity": 0.5,
    },
  },
  {
    "id": "Field Outline",
    "source": "example_source",
    "source-layer": "ftw-sources",
    "type": "line",
    "paint": {"line-color": "#ffffff", "line-width": 1, "line-opacity":
1},
  },
],
}

m.add_basemap("Esri.WorldImagery")
m.add_pmtiles(url, style=style, name="FTW", zoom_to_layer=False)
m.add_layer_control()
m

```




Figure 81: A map of agricultural fields.

22.13.3. 3D PMTiles

Render global building data in 3D for a realistic, textured experience. Set building colors and extrusion heights to create visually compelling cityscapes. For example, apply color gradients and height scaling based on building attributes to differentiate buildings by their heights.

```
url = "https://data.source.coop/cholmes/overture/overture-buildings.pmtiles"
metadata = leafmap.pmtiles_metadata(url)
print(f"layer names: {metadata['layer_names']}")
print(f"bounds: {metadata['bounds']}")
```

22.13.4. 3D Buildings

For a simplified setup, the `add_overture_3d_buildings` function quickly adds 3D building data from Overture's latest release to a basemap, creating depth and spatial realism on the map (see [Figure 82](#)).

```
m = leafmap.Map(
    center=[-74.0095, 40.7046], zoom=16, pitch=60, bearing=-17, style="positron",
    height="500px"
)
m.add_basemap("Esri.WorldImagery", visible=False)
m.add_overture_3d_buildings(template="simple")
m.add_layer_control()
m
```

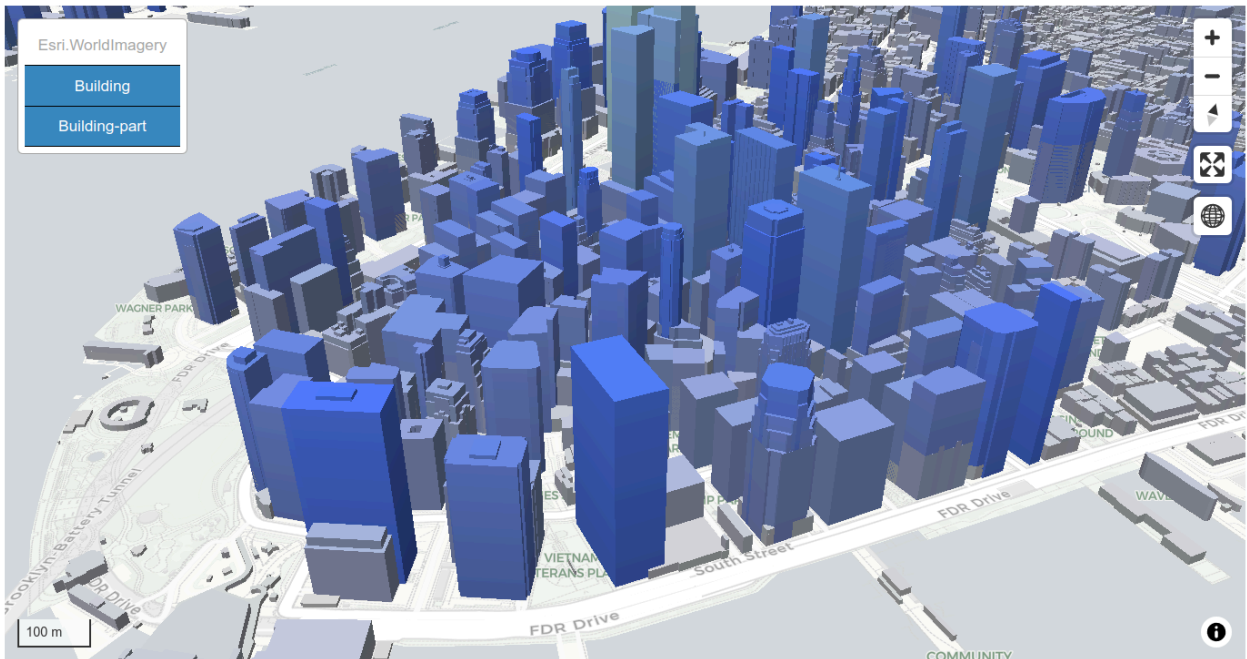



Figure 82: A map of New York City with Overture 3D buildings.

22.13.5. 2D Buildings, Transportation, and Other Themes

Switch between specific Overture themes, such as `buildings`, `transportation`, `places`, or `addresses`, depending on the data focus. Each theme overlays relevant features with adjustable opacity for thematic visualization, providing easy customization for urban planning or transportation studies.

Building theme:

```
m = leafmap.Map(center=[-74.0095, 40.7046], zoom=16)
m.add_basemap("Esri.WorldImagery", visible=False)
m.add_overture_data(theme="buildings", opacity=0.8)
m.add_layer_control()
m
```

Transportation theme:

```
m = leafmap.Map(center=[-74.0095, 40.7046], zoom=16)
m.add_basemap("Esri.WorldImagery", visible=False)
m.add_overture_data(theme="transportation", opacity=0.8)
m.add_layer_control()
m
```

Places theme:

```
m = leafmap.Map(center=[-74.0095, 40.7046], zoom=16)
m.add_basemap("Esri.WorldImagery", visible=False)
m.add_overture_data(theme="places", opacity=0.8)
m.add_layer_control()
m
```

Addresses theme:

```
m = leafmap.Map(center=[-74.0095, 40.7046], zoom=16)
m.add_basemap("Esri.WorldImagery", visible=False)
m.add_overture_data(theme="addresses", opacity=0.8)
m.add_layer_control()
m
```

Base theme:

```
m = leafmap.Map(center=[-74.0095, 40.7046], zoom=16)
m.add_basemap("Esri.WorldImagery", visible=False)
m.add_overture_data(theme="base", opacity=0.8)
m.add_layer_control()
m
```

Divisions theme:

```
m = leafmap.Map()
m.add_basemap("Esri.WorldImagery", visible=False)
m.add_overture_data(theme="divisions", opacity=0.8)
m.add_layer_control()
m
```

22.14. Adding DeckGL Layers

Integrate interactive, high-performance visualization layers using DeckGL. Leafmap's `add_deck_layer` method supports multiple DeckGL layer types, including Grid, GeoJSON, and Arc layers.

22.14.1. Single DeckGL Layer

Add a `GridLayer` to visualize point data, such as the density of bike parking in San Francisco. Customize the grid cell size, elevation, and color to represent data density visually and interactively.

```
m = leafmap.Map(
    style="positron",
    center=(-122.4, 37.74),
    zoom=12,
    pitch=40,
```

```

)
deck_grid_layer = {
  "@@type": "GridLayer",
  "id": "GridLayer",
  "data": "https://raw.githubusercontent.com/visgl/deck.gl/master/examples/
layer-browser/data/sf.bike.parking.json",
  "extruded": True,
  "getPosition": "@@=COORDINATES",
  "getColorWeight": "@@=SPACES",
  "getElevationWeight": "@@=SPACES",
  "elevationScale": 4,
  "cellSize": 200,
  "pickable": True,
}

m.add_deck_layers([deck_grid_layer], tooltip="Number of points: {{ count }}")
m

```

22.14.2. Multiple DeckGL Layers

Combine layers like `GeoJsonLayer` and `ArcLayer` for complex visualizations. For example:

1. Use `GeoJsonLayer` to show airports with varying point sizes based on significance.
2. Use `ArcLayer` to connect selected airports with London, coloring arcs to represent different paths (see [Figure 83](#)).

```

url = "https://d2ad6b4ur7yvpq.cloudfront.net/naturalearth-3.3.0/ne_10m_airports.
geojson"
data = leafmap.read_geojson(url)

```

```

m = leafmap.Map(
  style="positron",
  center=(0.45, 51.47),
  zoom=4,
  pitch=30,
)
deck_geojson_layer = {
  "@@type": "GeoJsonLayer",
  "id": "airports",
  "data": data,
  "filled": True,
  "pointRadiusMinPixels": 2,
  "pointRadiusScale": 2000,
  "getPointRadius": "@@=11 - properties.scalerank",
  "getFillColor": [200, 0, 80, 180],
  "autoHighlight": True,
}

```

```

    "pickable": True,
  }

  deck_arc_layer = {
    "@@type": "ArcLayer",
    "id": "arcs",
    "data": [
      feature
      for feature in data["features"]
      if feature["properties"]["scalerank"] < 4
    ],
    "getSourcePosition": [-0.4531566, 51.4709959], # London
    "getTargetPosition": "@@=geometry.coordinates",
    "getSourceColor": [0, 128, 200],
    "getTargetColor": [200, 0, 80],
    "getWidth": 2,
    "pickable": True,
  }

  m.add_deck_layers(
    [deck_geojson_layer, deck_arc_layer],
    tooltip={
      "airports": "{{ &properties.name }}",
      "arcs": "gps_code: {{ properties.gps_code }}",
    },
  )
m

```

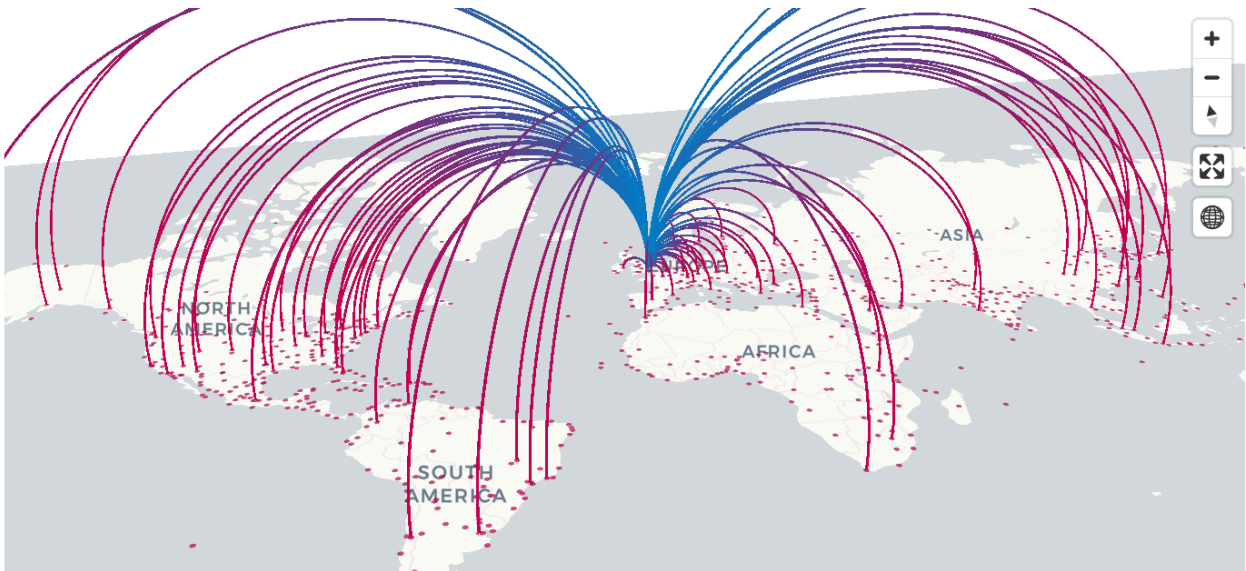


Figure 83: A map of airports with arcs between them.

The result is a rich, interactive visualization that highlights both point data and relational data connections, useful for airport connectivity or hub-and-spoke modeling.

22.15. Exporting to HTML

Export the final map to HTML using the `to_html` method. This allows you to share interactive maps online while keeping your private API keys secure. Create public API keys restricted to your website's domain for safety. Customize the HTML output, including dimensions, title, and embedding options.

```
# import os
# os.environ["MAPTILER_KEY"] = "YOUR_PRIVATE_API_KEY"
# os.environ["MAPTILER_KEY_PUBLIC"] = "YOUR_PUBLIC_API_KEY"
```

```
m = leafmap.Map(
    center=[-122.19861, 46.21168], zoom=13, pitch=60, bearing=150, style="3d-terrain"
)
m.add_layer_control(bg_layers=True)
m.to_html(
    "terrain.html",
    title="Awesome 3D Map",
    width="100%",
    height="100%",
    replace_key=True,
)
m
```

22.16. Key Takeaways

In this chapter, we explored the functionality of the MapLibre library for creating and customizing interactive maps in Python. We covered how to build a map from scratch, add controls, and manage different basemaps. Additionally, we explored more complex visualizations, including 3D building and terrain views, layer customization, and data integration with GeoJSON and raster formats. By understanding and applying these techniques, students are now equipped to develop dynamic geospatial visualizations using MapLibre, enhancing both analytical and presentation capabilities in their GIS projects.

22.17. Exercises

22.17.1. Exercise 1: Setting up MapLibre and Basic Map Creation

- Initialize a map centered on a country of your choice with an appropriate zoom level and display it with the `dark-matter` basemap.
- Change the basemap style to `liberty` and display it again.

22.17.2. Exercise 2: Customizing the Map View

- Create a 3D map of a city of your choice with an appropriate zoom level, pitch and bearing using the `liberty` basemap.
- Experiment with MapTiler 3D basemap styles, such as `3d-satellite`, `3d-hybrid`, and `3d-topo`, to visualize a location of your choice in different ways. Please set your MapTiler API key as Colab secret and do NOT expose the API key in the notebook.

22.17.3. Exercise 3: Adding Map Controls

- Create a map centered on a city of your choice and add the following controls to the map:
 - **Geolocate** control positioned at the top left.
 - **Fullscreen** control at the top right.
 - **Draw** control for adding points, lines, and polygons, positioned at the top left.

22.17.4. Exercise 4: Overlaying Data Layers

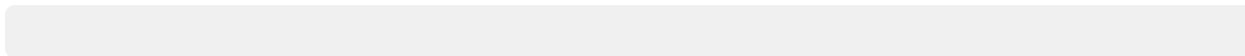
- **GeoJSON Layer:** Create a map and add the following GeoJSON data layers to the map with appropriate styles:
 - NYC buildings: https://github.com/opengeos/datasets/releases/download/places/nyc_buildings.geojson
 - NYC roads: https://github.com/opengeos/datasets/releases/download/places/nyc_roads.geojson
- **Thematic Raster Layer:** Create a map with a satellite basemap and add the following raster data layer to the map with an appropriate legend:
 - National Land Cover Database (NLCD) 2021: https://github.com/opengeos/datasets/releases/download/raster/nlcd_2021_land_cover_90m.tif
- **DEM Layer:** Create a map with a satellite basemap and add the following DEM layer to the map with an appropriate color bar:
 - DEM: <https://github.com/opengeos/datasets/releases/download/raster/dem.tif>
- **WMS Layer:** Create a map and add the ESA WorldCover WMS layer to the map with an appropriate legend:
 - url: <https://services.terrascope.be/wms/v2>
 - layers: `WORLDCOVER_2021_MAP`

22.17.5. Exercise 5: Working with 3D Buildings

- Set up a 3D map centered on a city of your choice with an appropriate zoom level, pitch, and bearing.
- Add 3D buildings to the map with extrusions based on their height attributes. Use a custom color gradient for the extrusion color.

22.17.6. Exercise 6: Adding Map Elements

- **Image and Text:** Add a logo image of your choice with appropriate text to the map.
- **GIF:** Add an animated GIF of your choice to the map.



Chapter 23. Cloud Computing with Earth Engine and Geemap

23.1. Introduction

This chapter introduces cloud-based geospatial analysis using [Google Earth Engine](https://earthengine.google.com)¹⁰⁸ (GEE) and the [geemap](https://geemap.org)¹⁰⁹ Python package. Google Earth Engine represents a paradigm shift from traditional desktop GIS to cloud computing, enabling efficient processing of massive datasets without local data storage or high-end hardware requirements.

The geemap package bridges Google Earth Engine’s power with Python’s familiar ecosystem. While Earth Engine traditionally used JavaScript, geemap enables direct access to Earth Engine capabilities from Jupyter notebooks, bringing planetary-scale geospatial analysis to Python users.

This chapter covers fundamental concepts and practical applications of geemap for geospatial analysis. You’ll learn to set up the library, work with raster and vector data, create interactive visualizations, and perform common geospatial operations—building a solid foundation for cloud-based geospatial analysis.

23.2. Learning Objectives

By the end of this chapter, you will be able to:

- Explain the fundamentals of the Google Earth Engine platform, including its data types and cloud-based capabilities
- Set up and configure the geemap library for conducting geospatial analyses in the cloud
- Perform essential geospatial operations, including filtering, visualizing, and exporting data from Earth Engine
- Access and manipulate both raster and vector data using Earth Engine
- Create timelapse animations from satellite imagery
- Create interactive charts from Earth Engine data

23.3. Introduction to Google Earth Engine

23.3.1. Google Earth Engine Overview

Google Earth Engine (GEE) is a cloud-computing platform that enables scientific analysis and visualization of large-scale geospatial datasets. It provides a rich data catalog and processing capabilities, allowing users to analyze satellite imagery and geospatial data at planetary scale.

Earth Engine is free for [noncommercial and research use](https://earthengine.google.com/noncommercial)¹¹⁰. Nonprofit organizations, research institutions, educators, Indigenous governments, and government researchers can continue using Earth Engine for free, as they have for more than a decade. However, [commercial users](https://earthengine.google.com/commercial)¹¹¹ may require a paid license.

¹⁰⁸<https://earthengine.google.com>

¹⁰⁹<https://geemap.org>

¹¹⁰<https://earthengine.google.com/noncommercial>

¹¹¹<https://earthengine.google.com/commercial>

23.3.2. Account Registration

Before using Google Earth Engine, you need a GEE account. Register at the [GEE Registration Page](https://code.earthengine.google.com/register)¹¹². During registration, you'll set up a Google Cloud Project and enable the Earth Engine API. Follow the instructions and note your Google Cloud Project ID for the next section.

23.3.3. Installing Geemap

The `geemap` package simplifies Google Earth Engine use in Python, providing an intuitive API for visualization and analysis in Jupyter notebooks. Geemap is pre-installed in Google Colab, but you may need additional dependencies for certain features. To install the latest version:

```
# %pip install -U geemap pygis
```

23.3.4. Import Libraries

To start, import the necessary libraries for working with Google Earth Engine (GEE) and `geemap`.

```
import ee
import geemap
```

23.3.5. Authenticate and Initialize Earth Engine

Before using Earth Engine's Python API, authenticate your identity and initialize a session. Authentication uses Cloud Projects for both free (noncommercial) and paid usage:

```
ee.Authenticate()
```

This opens a browser window for authentication. Follow the instructions and copy the authorization code if needed. Once authenticated, initialize a session with your Google Cloud Project ID:

```
ee.Initialize(project="your-ee-project")
```

23.3.6. Creating Interactive Maps

Let's create an interactive map using the `ipyleaflet` plotting backend. The `geemap.Map` class inherits from the `ipyleaflet.Map` class, so the syntax is similar to creating an interactive map with `ipyleaflet.Map`.

```
m = geemap.Map()
```

To display the map in a Jupyter notebook, simply enter the map object:

```
m
```

¹¹²<https://code.earthengine.google.com/register>

```
m
```

To center the map on the contiguous United States, use:

```
m = geemap.Map(center=[40, -100], zoom=4, height="600px")
m
```

To hide specific map controls, set the corresponding control argument to `False`, such as `draw_ctrl=False` to hide the drawing control.

```
m = geemap.Map(data_ctrl=False, toolbar_ctrl=False, draw_ctrl=False)
m
```

23.3.7. Adding Basemaps

There are several ways to add basemaps to a map in geemap. You can specify a basemap in the `basemap` keyword argument when creating the map, or you can add additional basemap layers using the `add_basemap` method. Geemap provides access to hundreds of built-in basemaps, making it easy to add layers to a map with a single line of code.

To create a map with a specific basemap, use the `basemap` argument as shown below. For example, `Esri.WorldImagery` provides an Esri world imagery basemap.

```
m = geemap.Map(basemap="Esri.WorldImagery")
m
```

23.3.7.1. Adding Multiple Basemaps

You can add multiple basemaps to a map by calling `add_basemap` multiple times. For example, the following code adds the `Esri.WorldTopoMap` and `OpenTopoMap` basemaps to the existing map:

```
m.add_basemap("Esri.WorldTopoMap")
```

```
m.add_basemap("OpenTopoMap")
```

23.3.8. Google Basemaps

Due to licensing restrictions, Google basemaps are not included in geemap by default. However, users can add Google basemaps manually at their own discretion using the following URLs:

```
ROADMAP: https://mt1.google.com/vt/lyrs=m&x={x}&y={y}&z={z}
SATELLITE: https://mt1.google.com/vt/lyrs=s&x={x}&y={y}&z={z}
```

```
TERRAIN: https://mt1.google.com/vt/lyrs=p&x={x}&y={y}&z={z}
HYBRID: https://mt1.google.com/vt/lyrs=y&x={x}&y={y}&z={z}
```

For example, to add Google Satellite as a tile layer:

```
m = geemap.Map()
url = "https://mt1.google.com/vt/lyrs=y&x={x}&y={y}&z={z}"
m.add_tile_layer(url, name="Google Satellite", attribution="Google")
m
```

You can also add text to the map using the `add_text` method. The text will be displayed at the specified location on the map.

```
m.add_text(text="Hello from Earth Engine", position="bottomright")
```

23.4. Introduction to Interactive Maps and Tools

23.4.1. Basemap Selector

The basemap selector allows you to choose from various basemaps via a dropdown menu. Adding it to your map provides easy access to different map backgrounds.

```
m = geemap.Map()
m.add("basemap_selector")
m
```

23.4.2. Layer Manager

The layer manager provides control over layer visibility and transparency. It enables toggling layers on and off and adjusting transparency with a slider, making it easy to customize map visuals.

```
m = geemap.Map(center=(40, -100), zoom=4)
dem = ee.Image("USGS/SRTMGL1_003")
states = ee.FeatureCollection("TIGER/2018/States")
vis_params = {
    "min": 0,
    "max": 4000,
    "palette": ["006633", "E5FFCC", "662A00", "D8D8D8", "F5F5F5"],
}
m.add_layer(dem, vis_params, "SRTM DEM")
m.add_layer(states, {}, "US States")
m.add("layer_manager")
m
```

23.4.3. Inspector Tool

The inspector tool allows you to click on the map to query Earth Engine data at specific locations. This is helpful for examining the properties of datasets directly on the map (see [Figure 84](#)).

```
m = geemap.Map(center=(40, -100), zoom=4)
dem = ee.Image("USGS/SRTMGL1_003")
landsat7 = ee.Image("LANDSAT/LE7_TOA_5YEAR/1999_2003")
states = ee.FeatureCollection("TIGER/2018/States")
vis_params = {
    "min": 0,
    "max": 4000,
    "palette": ["006633", "E5FFCC", "662A00", "D8D8D8", "F5F5F5"],
}
m.add_layer(dem, vis_params, "SRTM DEM")
m.add_layer(
    landsat7,
    {"bands": ["B4", "B3", "B2"], "min": 20, "max": 200, "gamma": 2.0},
    "Landsat 7",
)
m.add_layer(states, {}, "US States")
m.add("inspector")
m
```

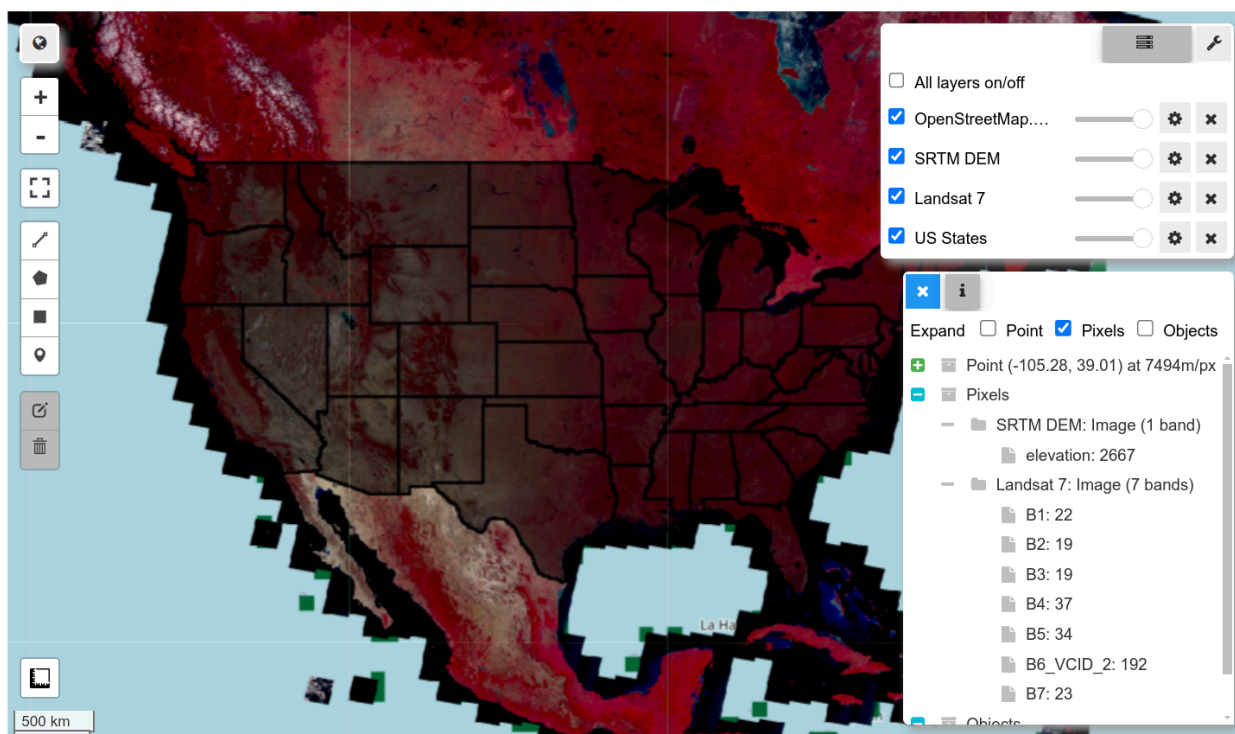


Figure 84: The geemap inspector for querying Earth Engine data.

23.4.4. Layer Editor

With the layer editor, you can adjust visualization parameters of Earth Engine data for better clarity and focus. It supports single-band images, multi-band images, and feature collections.

23.4.4.1. Single-Band image

The example below shows how to use the layer editor to adjust the visualization parameters of a single-band image (Figure 85). Select a colormap from the dropdown menu, and then adjust the min and max values. Click the “Apply” button to update the visualization.

```
m = geemap.Map(center=(40, -100), zoom=4)
dem = ee.Image("USGS/SRTMGL1_003")
vis_params = {
    "min": 0,
    "max": 4000,
    "palette": ["006633", "E5FFCC", "662A00", "D8D8D8", "F5F5F5"],
}
m.add_layer(dem, vis_params, "SRTM DEM")
m.add("layer_editor", layer_dict=m.ee_layers["SRTM DEM"])
m
```

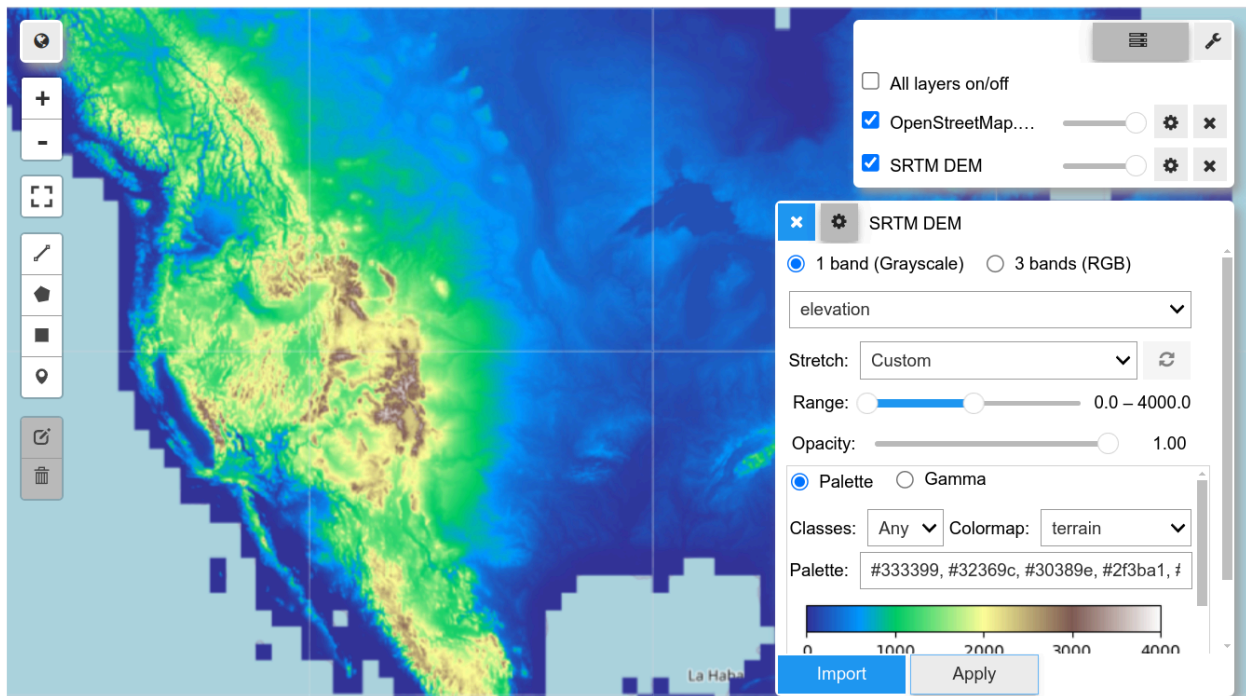


Figure 85: Layer Editor for a single band image.

23.4.4.2. Multi-Band image

The example below shows how to use the layer editor to adjust the visualization parameters of a multi-band image (Figure 86). Select an RGB band combination from the dropdown menu, and then adjust the min and max values. Click the “Apply” button to update the visualization.

```
m = geemap.Map(center=(40, -100), zoom=4)
landsat7 = ee.Image("LANDSAT/LE7_TOA_5YEAR/1999_2003")
m.add_layer(
    landsat7,
    {"bands": ["B4", "B3", "B2"], "min": 20, "max": 200, "gamma": 2.0},
    "Landsat 7",
)
m.add("layer_editor", layer_dict=m.ee_layers["Landsat 7"])
m
```

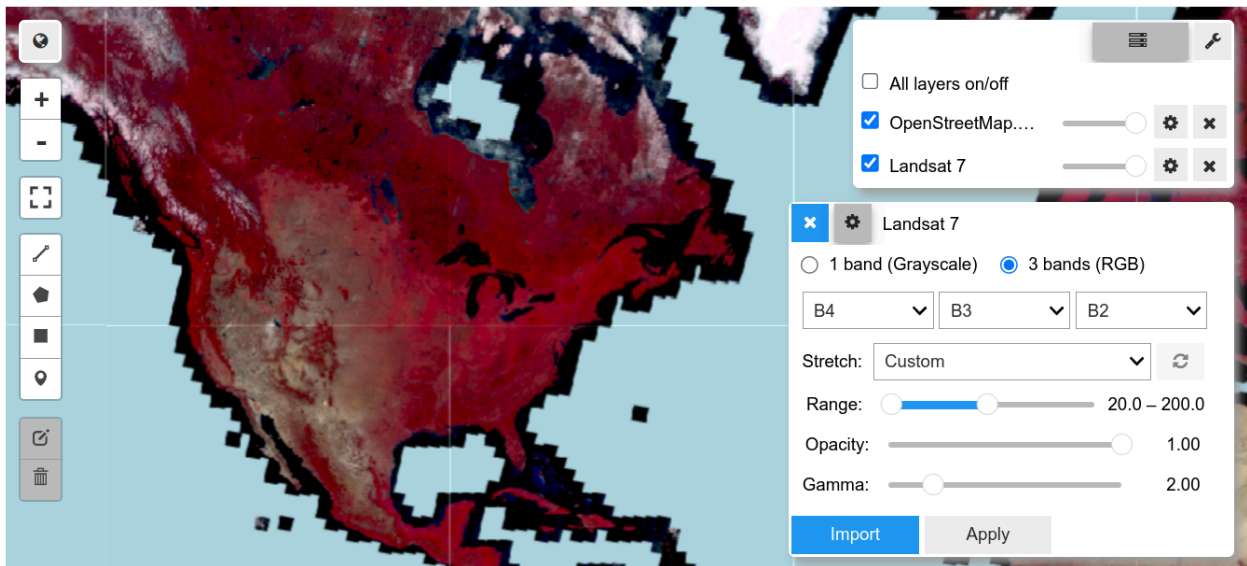


Figure 86: Layer Editor for a multi-band image.

23.4.4.3. Feature Collection

The example below shows how to use the layer editor to adjust the visualization parameters of a feature collection (Figure 87). Adjust the color, opacity, and line width. Click the “Apply” button to update the visualization.

```
m = geemap.Map(center=(40, -100), zoom=4)
states = ee.FeatureCollection("TIGER/2018/States")
m.add_layer(states, {}, "US States")
m.add("layer_editor", layer_dict=m.ee_layers["US States"])
m
```

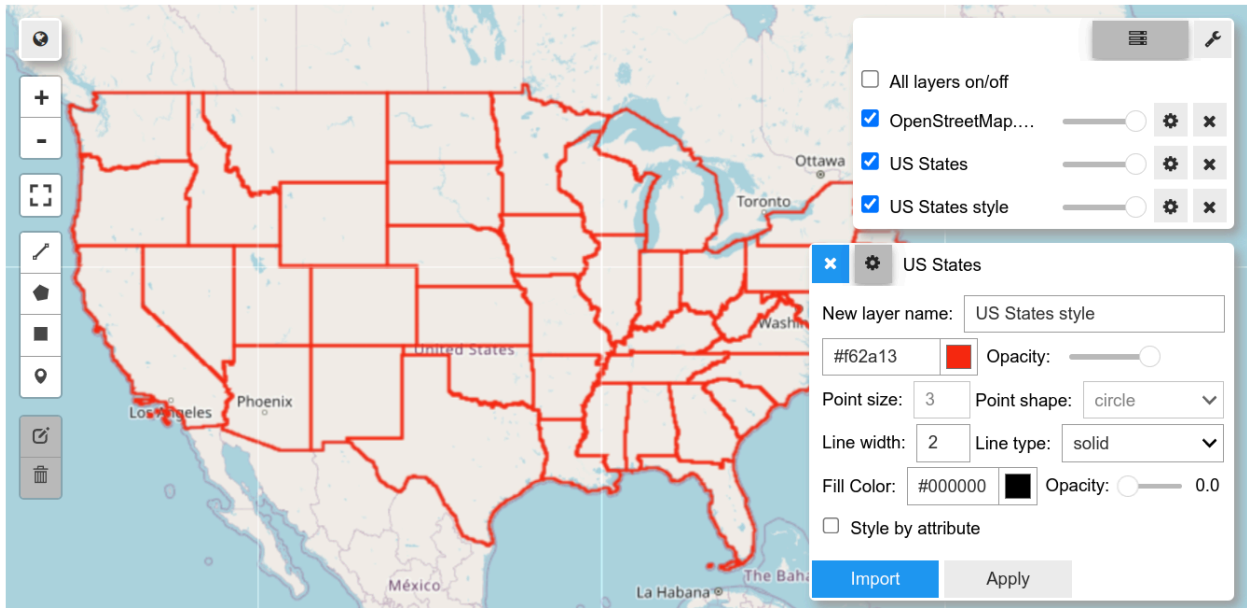


Figure 87: Layer Editor for a feature collection.

23.4.5. Draw Control

The draw control feature allows you to draw shapes directly on the map, converting them automatically into Earth Engine objects. Access the drawn features as follows:

- To return the last drawn feature as an `ee.Geometry()`, use `m.user_roi`
- To return all drawn features as an `ee.FeatureCollection()`, use `m.user_rois`

```
m = geemap.Map(center=(40, -100), zoom=4)
dem = ee.Image("USGS/SRTMGL1_003")
vis_params = {
    "min": 0,
    "max": 4000,
    "palette": "terrain",
}
m.add_layer(dem, vis_params, "SRTM DEM")
m.add("layer_manager")
m
```

Use the drawing tools to draw shapes on the map. Then run the code below to clip the DEM to the drawn shape.

```
if m.user_roi is not None:
    image = dem.clip(m.user_roi)
    m.layers[1].visible = False
    m.add_layer(image, vis_params, "Clipped DEM")
```


23.5. The Earth Engine Data Catalog

The [Earth Engine Data Catalog](https://developers.google.com/earth-engine/datasets)¹¹³ hosts an extensive collection of geospatial datasets. Currently, the catalog includes over 1,000 datasets with a combined size exceeding 100 petabytes. Notable datasets include Landsat, Sentinel, MODIS, and NAIP. For a comprehensive list of datasets in CSV or JSON format, refer to the [Earth Engine Datasets List](https://developers.google.com/earth-engine/datasets/catalog)¹¹⁴.

23.5.1. Searching Datasets on the Earth Engine Website

To browse datasets directly on the Earth Engine website:

- View the full catalog: <https://developers.google.com/earth-engine/datasets/catalog>
- Search by tags: <https://developers.google.com/earth-engine/datasets/tags>

23.5.2. Searching Datasets Within Geemap

The Earth Engine Data Catalog can also be searched directly from geemap. Click on the globe icon in the upper left corner to open the data search panel. Click on the **data** tab to search for datasets (see [Figure 88](#)). Enter keywords to filter datasets by name, tag, or description. For instance, searching for “elevation” will display only datasets containing “elevation” in their metadata, returning 52 datasets. You can scroll through the results to locate the [NASA SRTM Digital Elevation 30m](https://developers.google.com/earth-engine/datasets/catalog/USGS_SRTMGL1_003#description)¹¹⁵ dataset.

```
m = geemap.Map()  
m
```

¹¹³<https://developers.google.com/earth-engine/datasets>

¹¹⁴<https://tinyurl.com/gee-catalog>

¹¹⁵https://developers.google.com/earth-engine/datasets/catalog/USGS_SRTMGL1_003#description

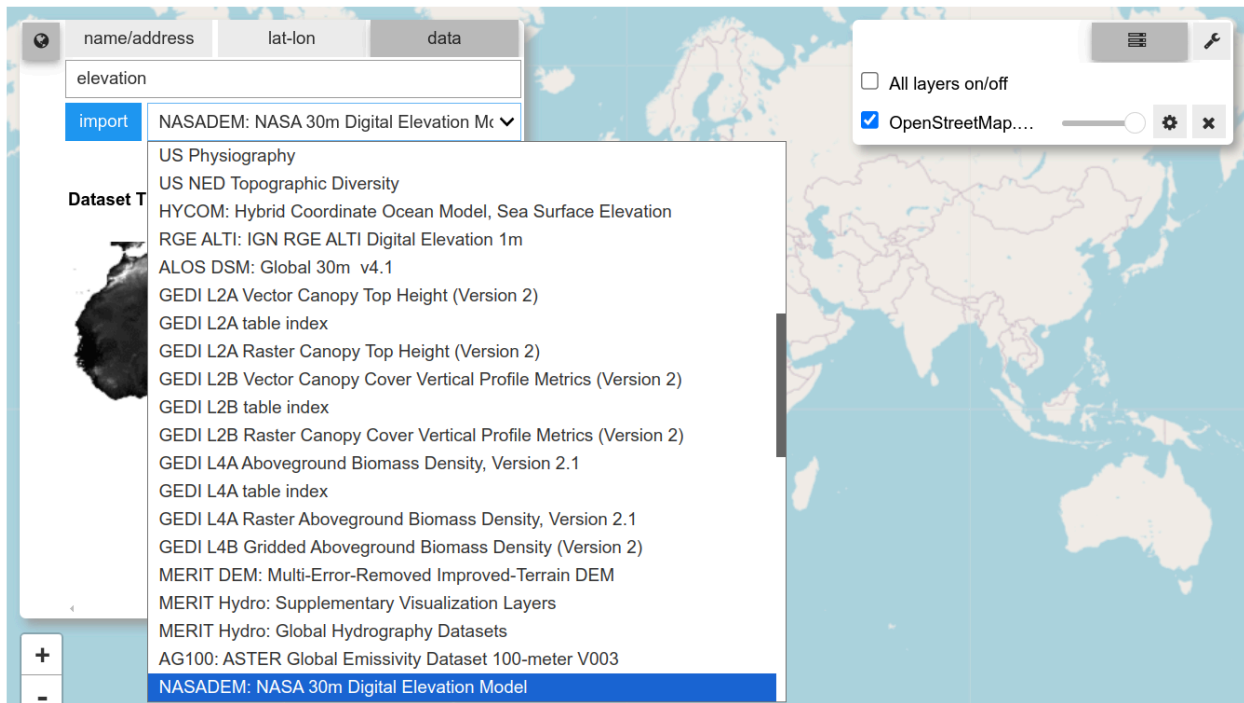


Figure 88: Search for datasets in the data search panel.

Each dataset page contains detailed information such as availability, provider, Earth Engine snippet, tags, description, and example code. The Image, ImageCollection, or FeatureCollection ID is essential for accessing the dataset in Earth Engine's JavaScript or Python APIs.

Click in the blue **import** button to copy the code snippet to your clipboard. Then paste the code snippet into your Jupyter notebook, which you can modify as needed.

23.5.3. Using the Datasets Module in Geemap

Geemap offers a built-in `datasets` module to access specific datasets programmatically. For example, to access the USGS GAP Alaska dataset:

```
from geemap.datasets import DATA
```

```
m = geemap.Map()
dataset = ee.Image(DATA.USGS_GAP_AK_2001)
m.add_layer(dataset, {}, "GAP Alaska")
m.centerObject(dataset, zoom=4)
m
```

To retrieve metadata for a specific dataset, use the `get_metadata` function:

```
from geemap.datasets import get_metadata
```

```
get_metadata(DATA.USGS_GAP_AK_2001)
```

23.6. Earth Engine Data Types

Earth Engine objects are server-side entities, meaning they are stored and processed on Earth Engine's servers rather than locally. This setup is comparable to streaming services (e.g., YouTube or Netflix) where the data remains on the provider's servers, allowing us to access and process geospatial data in real time without downloading it to our local devices.

- **Image**: The core raster data type in Earth Engine, representing single images or scenes.
- **ImageCollection**: A collection or sequence of images, often used for time series analysis.
- **Geometry**: The fundamental vector data type, including shapes like points, lines, and polygons.
- **Feature**: A Geometry with associated attributes, used to add descriptive data to vector shapes.
- **FeatureCollection**: A collection of Features, similar to a shapefile with attribute data.

23.7. Earth Engine Raster Data

23.7.1. Image

In Earth Engine, raster data is represented as **Image** objects. Each Image is composed of bands, with each band having its own name, data type, scale, mask, and projection. Metadata for each image is stored as a set of properties.

23.7.1.1. Loading Earth Engine Images

To load images, use the Earth Engine asset ID within the `ee.Image` constructor. Asset IDs can be found in the Earth Engine Data Catalog. For example, to load the NASA SRTM Digital Elevation dataset:

```
image = ee.Image("USGS/SRTMGL1_003")
image
```

```
▼ Image USGS/SRTMGL1_003 (1 band)
  type: Image
  id: USGS/SRTMGL1_003
  version: 1641990767055141
  ▼ bands: List (1 element)
    ▼ 0: "elevation", signed int16, EPSG:4326, 1296001x417601 px
      id: elevation
      crs: EPSG:4326
      ► crs_transform: List (6 elements)
      ► data_type: signed int16
      ► dimensions: [1296001, 417601]
    ► properties: Object (24 properties)
```

23.7.1.2. Visualizing Earth Engine Images

To visualize an image, you can specify visualization parameters such as minimum and maximum values and color palettes.

```
m = geemap.Map(center=[21.79, 70.87], zoom=3)
image = ee.Image("USGS/SRTMGL1_003")
vis_params = {
    "min": 0,
    "max": 6000,
    "palette": ["006633", "E5FFCC", "662A00", "D8D8D8", "F5F5F5"], # 'terrain'
}
m.add_layer(image, vis_params, "SRTM")
m
```

23.7.2. ImageCollection

An **ImageCollection** represents a sequence of images, often used for temporal data like satellite image time series. ImageCollections are created by passing an Earth Engine asset ID into the `ImageCollection` constructor. Asset IDs are available in the [Earth Engine Data Catalog](#).

23.7.2.1. Loading Image Collections

To load an ImageCollection, such as the Sentinel-2 surface reflectance collection:

```
collection = ee.ImageCollection("COPERNICUS/S2_SR_HARMONIZED")
```

23.7.2.2. Filtering Image Collections

You can filter ImageCollections by location and time. For example, to filter images covering a specific location with low cloud cover:

```
geometry = ee.Geometry.Point([-83.909463, 35.959111])
images = collection.filterBounds(geometry)
images.size()
```

Let's check the metadata of the first image in the collection:

```
images.first()
```

We can also filter the collection by location, date, and cloud cover:

```
images = (
    collection.filterBounds(geometry)
    .filterDate("2024-07-01", "2024-10-01")
    .filter(ee.Filter.lt("CLOUDY_PIXEL_PERCENTAGE", 5))
)
```

```
)  
images.size()
```

To view the filtered collection on a map:

```
m = geemap.Map()  
image = images.first()  
  
vis = {  
    "min": 0.0,  
    "max": 3000,  
    "bands": ["B4", "B3", "B2"],  
}  
  
m.add_layer(image, vis, "Sentinel-2")  
m.center_object(image, 8)  
m
```

23.7.2.3. Visualizing Image Collections

To visualize an **ImageCollection** as a single composite image, you need to aggregate the collection. For example, using the `collection.median()` method creates an image representing the median value across the collection.

```
m = geemap.Map()  
image = images.median()  
  
vis = {  
    "min": 0.0,  
    "max": 3000,  
    "bands": ["B8", "B4", "B3"],  
}  
  
m.add_layer(image, vis, "Sentinel-2")  
m.center_object(geometry, 8)  
m
```

23.8. Earth Engine Vector Data

A **FeatureCollection** is a collection of Features. It functions similarly to a GeoJSON FeatureCollection, where features include associated properties or attributes. For example, a shapefile's data can be represented as a FeatureCollection.

23.8.1. Loading Feature Collections

The [Earth Engine Data Catalog](#) includes various vector datasets, such as U.S. Census data and country boundaries, as feature collections. Feature Collection IDs are accessible by searching the data catalog. For example, to load the [TIGER roads data](#)¹¹⁶ by the U.S. Census Bureau:

```
m = geemap.Map()
fc = ee.FeatureCollection("TIGER/2016/Roads")
m.set_center(-83.909463, 35.959111, 12)
m.add_layer(fc, {}, "Census roads")
m
```

23.8.2. Filtering Feature Collections

The `filter` method allows you to filter a `FeatureCollection` based on certain attribute values. For instance, we can filter for specific states using the `eq` filter to select “Tennessee”:

```
m = geemap.Map()
states = ee.FeatureCollection("TIGER/2018/States")
fc = states.filter(ee.Filter.eq("NAME", "Tennessee"))
m.add_layer(fc, {}, "Tennessee")
m.center_object(fc, 7)
m
```

To retrieve properties of the first feature in the collection, use:

```
feat = fc.first()
feat.toDictionary()
```

You can also convert a `FeatureCollection` to a `Pandas DataFrame` for easier analysis with:

```
geemap.ee_to_df(fc)
```

The `ee.Filter.inList()` method is used to filter features based on a list of values. For example, to filter the US States dataset to only include California, Oregon, and Washington, we can use the following code:

```
m = geemap.Map()
states = ee.FeatureCollection("TIGER/2018/States")
fc = states.filter(ee.Filter.inList("NAME", ["California", "Oregon",
"Washington"]))
m.add_layer(fc, {}, "West Coast")
m.center_object(fc, 5)
m
```

¹¹⁶<https://tinyurl.com/gee-tiger-roads>

The `filterBounds()` method is used to filter a feature collection by a geometry. For example, to filter the US states that intersect with a region of interest:

```
region = m.user_roi
if region is None:
    region = ee.Geometry.BBox(-88.40, 29.88, -77.90, 35.39)

fc = ee.FeatureCollection("TIGER/2018/States").filterBounds(region)
m.add_layer(fc, {}, "Southeastern U.S.")
m.center_object(fc, 6)
```

A FeatureCollection can be used to clip an image. The `clipToCollection()` method clips an image to the geometry of a feature collection. For example, to clip a Landsat image to the boundary of Germany:

```
m = geemap.Map(center=(40, -100), zoom=4)
landsat7 = ee.Image("LANDSAT/LE7_TOA_5YEAR/1999_2003")
countries = ee.FeatureCollection("FAO/GAUL_SIMPLIFIED_500m/2015/level0")
fc = countries.filter(ee.Filter.eq("ADM0_NAME", "Germany"))
image = landsat7.clipToCollection(fc)
m.add_layer(
    image,
    {"bands": ["B4", "B3", "B2"], "min": 20, "max": 200, "gamma": 2.0},
    "Landsat 7",
)
m.center_object(fc, 6)
m
```

23.8.3. Visualizing Feature Collections

Once loaded, feature collections can be visualized on an interactive map. For example, visualizing U.S. state boundaries from the census data:

```
m = geemap.Map(center=[40, -100], zoom=4)
states = ee.FeatureCollection("TIGER/2018/States")
m.add_layer(states, {}, "US States")
m
```

Feature collections can also be styled with additional parameters. To apply a custom style, specify options like color and line width:

```
m = geemap.Map(center=[40, -100], zoom=4)
states = ee.FeatureCollection("TIGER/2018/States")
style = {"color": "0000ffff", "width": 2, "lineType": "solid", "fillColor":
"FF000080"}
m.add_layer(states.style(**style), {}, "US States")
m
```

Using `Map.add_styled_vector()` method, you can apply a color palette to style different features by attribute:

```
m = geemap.Map(center=[40, -100], zoom=4)
states = ee.FeatureCollection("TIGER/2018/States")
vis_params = {
    "color": "000000",
    "colorOpacity": 1,
    "pointSize": 3,
    "pointShape": "circle",
    "width": 2,
    "lineType": "solid",
    "fillColorOpacity": 0.66,
}
palette = ["006633", "E5FFCC", "662A00", "D8D8D8", "F5F5F5"]
m.add_styled_vector(
    states, column="NAME", palette=palette, layer_name="Styled vector",
    **vis_params
)
m
```

23.9. More Tools for Visualizing Earth Engine Data

23.9.1. Using the Plotting Tool

The plotting tool in geemap enables visualization of Earth Engine data layers. In this example, Landsat 7 and Hyperion data are added with specific visualization parameters, allowing for comparison of these datasets. The plot GUI is then added to facilitate detailed exploration of the data ([Figure 89](#)).

```
m = geemap.Map(center=[40, -100], zoom=4)

landsat7 = ee.Image("LANDSAT/LE7_TOA_5YEAR/1999_2003").select(
    ["B1", "B2", "B3", "B4", "B5", "B7"]
)

landsat_vis = {"bands": ["B4", "B3", "B2"], "gamma": 1.4}
m.add_layer(landsat7, landsat_vis, "Landsat")

hyperion = ee.ImageCollection("E01/HYPERION").filter(
    ee.Filter.date("2016-01-01", "2017-03-01")
)

hyperion_vis = {
    "min": 1000.0,
    "max": 14000.0,
    "gamma": 2.5,
}
```

```
m.add_layer(hyperion, hyperion_vis, "Hyperion")
m.add_plot_gui()
m
```

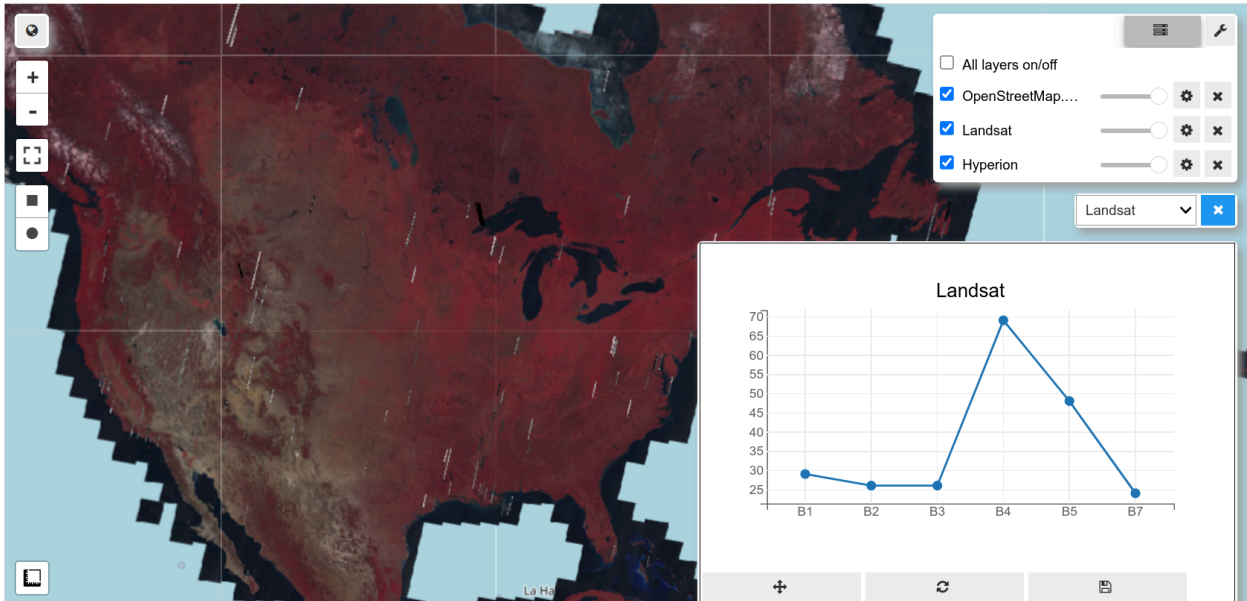


Figure 89: Plotting the spectral signature of a Landsatpixel.

Set the plotting options for Landsat to add marker clusters and overlays, enhancing the interactivity of plotted data on the map.

```
m.set_plot_options(add_marker_cluster=True, overlay=True)
```

Adjust the plotting options for Hyperion data by setting a bar plot type with marker clusters, suitable for visualizing Hyperion's data values in bar format.

```
m.set_plot_options(add_marker_cluster=True, plot_type="bar")
```

23.9.2. Legends

23.9.2.1. Built-in Legends

Geemap provides built-in legends that can be easily added to maps for better interpretability of data classes. To check all available built-in legends, run the code below:

```
from geemap.legends import builtin_legends

for legend in builtin_legends:
    print(legend)
```


Add an NLCD WMS layer along with its corresponding legend, which appears as an overlay, providing users with an informative legend display.

```
m = geemap.Map(center=[40, -100], zoom=4)
m.add_basemap("Esri.WorldImagery")
m.add_basemap("NLCD 2021 CONUS Land Cover")
m.add_legend(builtin_legend="NLCD", max_width="100px", height="455px")
m
```

Add an Earth Engine layer for NLCD land cover and display its legend, specifying title, legend type, and dimensions for user convenience (Figure 90).

```
m = geemap.Map(center=[40, -100], zoom=4)
m.add_basemap("Esri.WorldImagery")

nlcd = ee.Image("USGS/NLCD_RELEASES/2021_REL/NLCD/2021")
landcover = nlcd.select("landcover")

m.add_layer(landcover, {}, "NLCD Land Cover 2021")
m.add_legend(
    title="NLCD Land Cover Classification", builtin_legend="NLCD", height="455px"
)
m
```

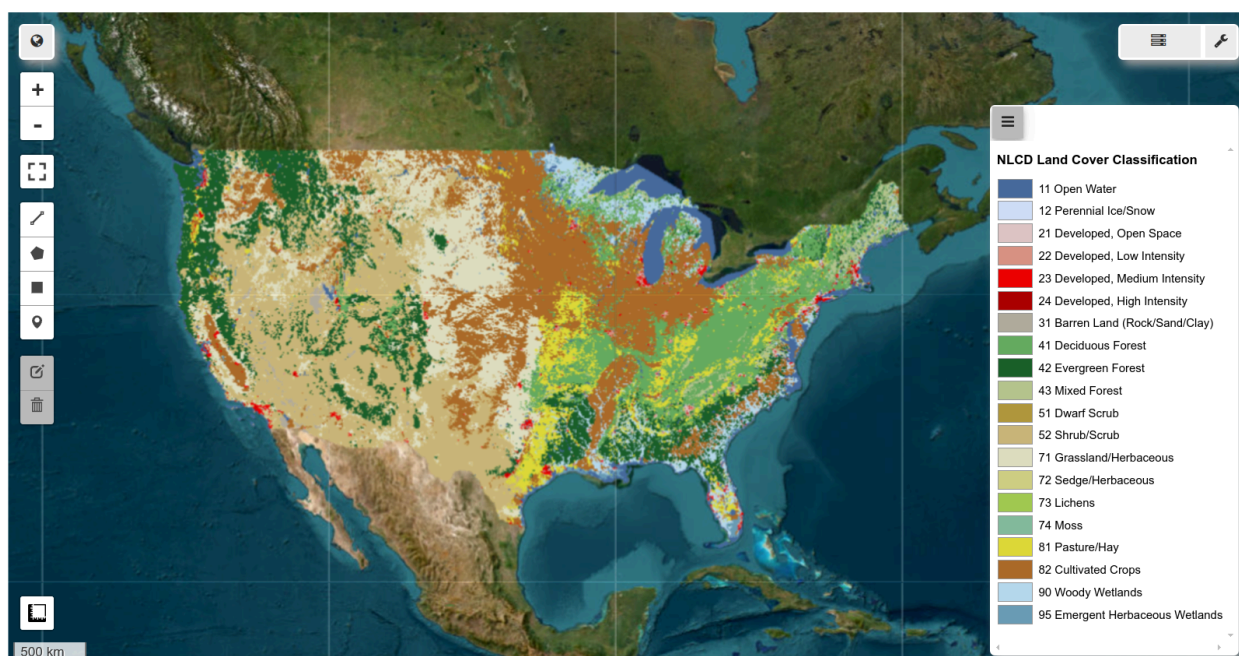


Figure 90: A legend for the NLCD land cover dataset.

23.9.2.2. Custom Legends

Create a custom legend by defining unique labels and colors for each class, allowing for flexible map customization. Define a custom legend using a dictionary that links specific colors to labels. This approach provides flexibility to represent specific categories in the data, such as various land cover types.

```
m = geemap.Map(center=[40, -100], zoom=4)
m.add_basemap("Esri.WorldImagery")

legend_dict = {
    "11 Open Water": "466b9f",
    "12 Perennial Ice/Snow": "d1def8",
    "21 Developed, Open Space": "dec5c5",
    "22 Developed, Low Intensity": "d99282",
    "23 Developed, Medium Intensity": "eb0000",
    "24 Developed High Intensity": "ab0000",
    "31 Barren Land (Rock/Sand/Clay)": "b3ac9f",
    "41 Deciduous Forest": "68ab5f",
    "42 Evergreen Forest": "1c5f2c",
    "43 Mixed Forest": "b5c58f",
    "51 Dwarf Scrub": "af963c",
    "52 Shrub/Scrub": "ccb879",
    "71 Grassland/Herbaceous": "dfdfc2",
    "72 Sedge/Herbaceous": "d1d182",
    "73 Lichens": "a3cc51",
    "74 Moss": "82ba9e",
    "81 Pasture/Hay": "dcd939",
    "82 Cultivated Crops": "ab6c28",
    "90 Woody Wetlands": "b8d9eb",
    "95 Emergent Herbaceous Wetlands": "6c9fb8",
}

nlcd = ee.Image("USGS/NLCD_RELEASES/2021_REL/NLCD/2021")
landcover = nlcd.select("landcover")

m.add_layer(landcover, {}, "NLCD Land Cover 2021")
m.add_legend(title="NLCD Land Cover Classification", legend_dict=legend_dict)
m
```

23.9.3. Color Bars

Add a horizontal color bar representing elevation data. This example demonstrates how to add an SRTM elevation layer and overlay a color bar for quick visual reference of elevation values ([Figure 91](#)).

```
m = geemap.Map()

dem = ee.Image("USGS/SRTMGL1_003")
vis_params = {
```

```

    "min": 0,
    "max": 4000,
    "palette": ["006633", "E5FFCC", "662A00", "D8D8D8", "F5F5F5"],
}

m.add_layer(dem, vis_params, "SRTM DEM")
m.add_colorbar(vis_params, label="Elevation (m)", layer_name="SRTM DEM")
m

```

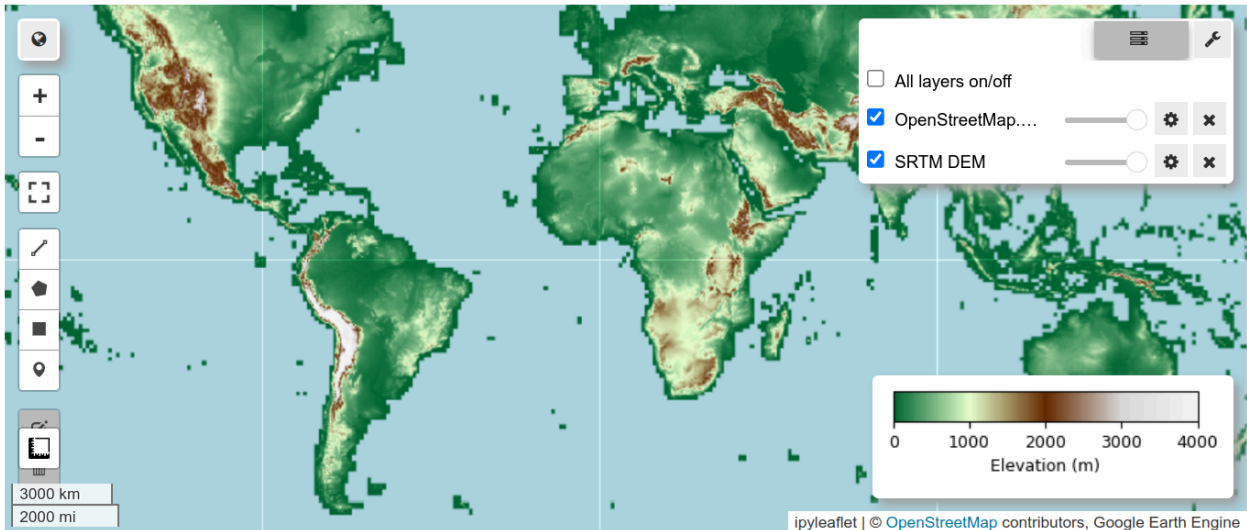


Figure 91: Adding A colorbar for a DEM.

Add a vertical color bar for elevation data, adjusting its orientation and dimensions to fit the map layout.

```

m.add_colorbar(
    vis_params,
    label="Elevation (m)",
    layer_name="SRTM DEM",
    orientation="vertical",
    max_width="100px",
)

```

23.9.4. Split-panel Maps

Create a split-panel map with basemaps, allowing users to compare two different basemaps side by side.

```

m = geemap.Map()
m.split_map(left_layer="Esri.WorldTopoMap", right_layer="OpenTopoMap")
m

```

Use Earth Engine layers in a split-panel map to compare NLCD data from 2001 and 2021. This layout is effective for examining changes between datasets across time.

```

m = geemap.Map(center=(40, -100), zoom=4, height="600px")

nlcd_2001 = ee.Image("USGS/NLCD_RELEASES/2019_REL/NLCD/2001").select("landcover")
nlcd_2021 = ee.Image("USGS/NLCD_RELEASES/2021_REL/NLCD/2021").select("landcover")

left_layer = geemap.ee_tile_layer(nlcd_2001, {}, "NLCD 2001")
right_layer = geemap.ee_tile_layer(nlcd_2021, {}, "NLCD 2021")

m.split_map(left_layer, right_layer)
m

```

23.9.5. Linked Maps

Set up a 2x2 grid of linked maps showing Sentinel-2 imagery in different band combinations, ideal for comparing multiple visual perspectives (see [Figure 92](#)). Note that this feature may not work in Colab.

```

image = (
    ee.ImageCollection("COPERNICUS/S2_SR_HARMONIZED")
    .filterDate("2024-07-01", "2024-10-01")
    .filter(ee.Filter.lt("CLOUDY_PIXEL_PERCENTAGE", 5))
    .map(lambda img: img.divide(10000))
    .median()
)

vis_params = [
    {"bands": ["B4", "B3", "B2"], "min": 0, "max": 0.3, "gamma": 1.3},
    {"bands": ["B8", "B11", "B4"], "min": 0, "max": 0.3, "gamma": 1.3},
    {"bands": ["B8", "B4", "B3"], "min": 0, "max": 0.3, "gamma": 1.3},
    {"bands": ["B12", "B12", "B4"], "min": 0, "max": 0.3, "gamma": 1.3},
]

labels = [
    "Natural Color (B4/B3/B2)",
    "Land/Water (B8/B11/B4)",
    "Color Infrared (B8/B4/B3)",
    "Vegetation (B12/B11/B4)",
]

geemap.linked_maps(
    rows=2,
    cols=2,
    height="300px",
    center=[35.959111, -83.909463],
    zoom=12,
    ee_objects=[image],
    vis_params=vis_params,
    labels=labels,
)

```



```
label_position="topright",
)
```

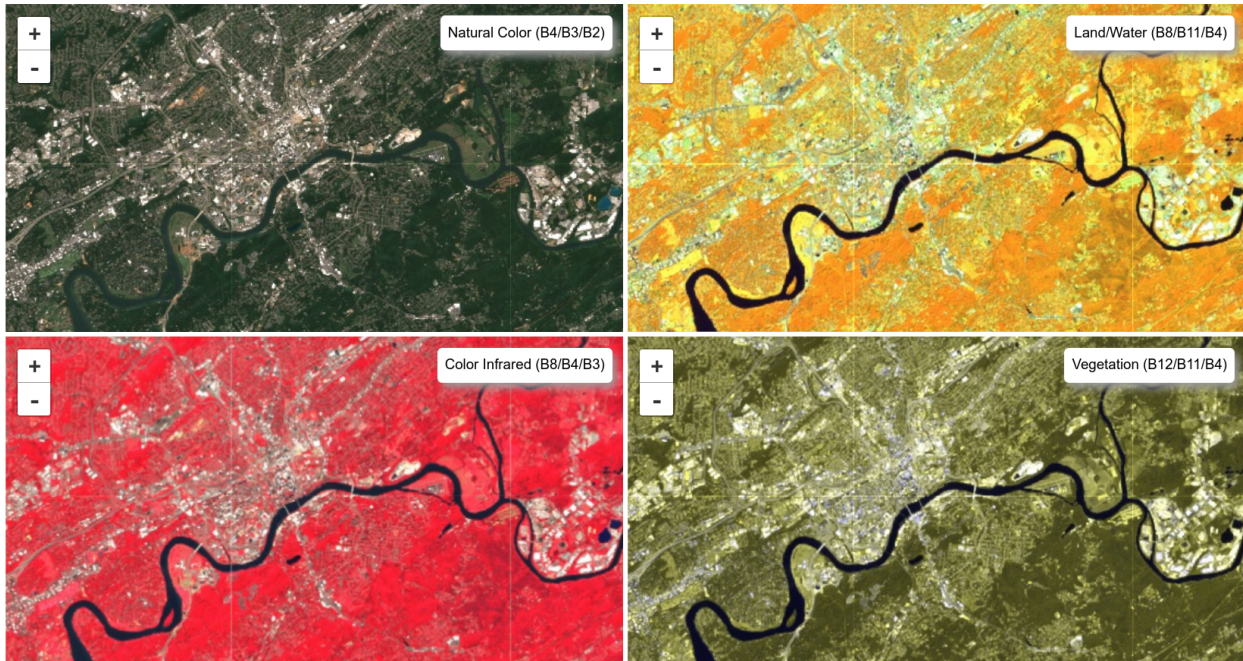


Figure 92: Linked maps showing Sentinel-2 imagery in different band combinations.

23.9.6. Timeseries Inspector

Retrieve the available years in the NLCD collection for setting up a timeseries. This step helps confirm available time points for inspecting changes over time.

```
m = geemap.Map(center=[40, -100], zoom=4)
collection = ee.ImageCollection("USGS/NLCD_RELEASES/2019_REL/
NLCD").select("landcover")
vis_params = {"bands": ["landcover"]}
years = collection.aggregate_array("system:index").getInfo()
years
```

Use a timeseries inspector to compare changes in NLCD data across different years. This tool is ideal for temporal data visualization, showing how land cover changes in an interactive format (see [Figure 93](#)).

```
m.ts_inspector(
    left_ts=collection,
    right_ts=collection,
    left_names=years,
    right_names=years,
    left_vis=vis_params,
```

```

    right_vis=vis_params,
    width="80px",
  )
m

```

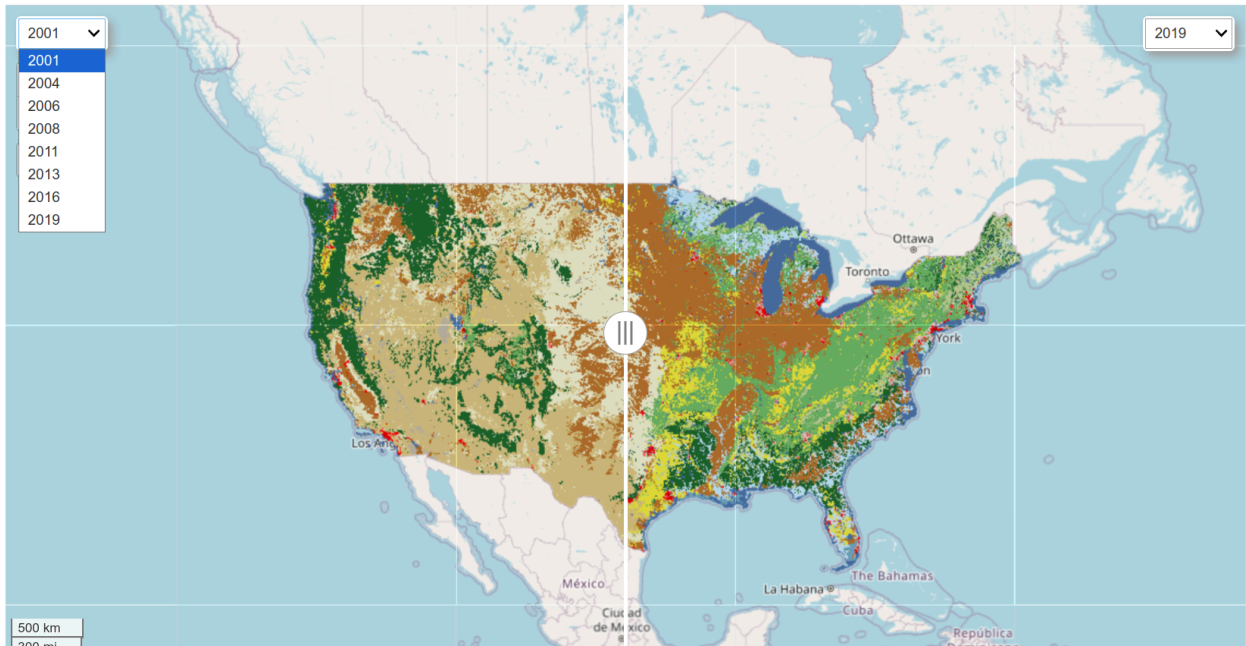


Figure 93: A timeseries inspector for visualizing time series data.

23.9.7. Time Slider

Create a time slider to explore MODIS vegetation data, setting the slider to visualize NDVI values over a specific period. This feature allows users to track vegetation changes month-by-month.

```

m = geemap.Map()

collection = (
    ee.ImageCollection("MODIS/MCD43A4_006_NDVI")
    .filter(ee.Filter.date("2018-06-01", "2018-07-01"))
    .select("NDVI")
)
vis_params = {
    "min": 0.0,
    "max": 1.0,
    "palette": "ndvi",
}

m.add_time_slider(collection, vis_params, time_interval=2)
m

```

Add a time slider for visualizing NOAA weather data over a 24-hour period, using color-coding to show temperature variations. The slider enables temporal exploration of hourly data.

```
m = geemap.Map()

collection = (
    ee.ImageCollection("NOAA/GFS0P25")
    .filterDate("2018-12-22", "2018-12-23")
    .limit(24)
    .select("temperature_2m_above_ground")
)

vis_params = {
    "min": -40.0,
    "max": 35.0,
    "palette": ["blue", "purple", "cyan", "green", "yellow", "red"],
}

labels = [str(n).zfill(2) + ":00" for n in range(0, 24)]
m.add_time_slider(collection, vis_params, labels=labels, time_interval=1,
opacity=0.8)
m
```

Add a time slider to visualize Sentinel-2 imagery with specific bands and cloud cover filtering. This feature enables temporal analysis of imagery data, allowing users to explore seasonal and other changes over time.

```
m = geemap.Map()

collection = (
    ee.ImageCollection("COPERNICUS/S2_SR_HARMONIZED")
    .filterBounds(ee.Geometry.Point([-83.909463, 35.959111]))
    .filterMetadata("CLOUDY_PIXEL_PERCENTAGE", "less_than", 10)
    .filter(ee.Filter.calendarRange(6, 8, "month"))
)

vis_params = {"min": 0, "max": 4000, "bands": ["B8", "B4", "B3"]}

m.add_time_slider(collection, vis_params)
m.set_center(-83.909463, 35.959111, 12)
m
```

23.10. Vector Data Processing

23.10.1. From GeoJSON

Load GeoJSON data into an Earth Engine FeatureCollection. This example retrieves countries data from a remote URL, converting it to a FeatureCollection and visualizing it with a specified style.

```

in_geojson = "https://github.com/gee-community/geemap/blob/master/examples/data/
countries.geojson"
m = geemap.Map()
fc = geemap.geojson_to_ee(in_geojson)
m.add_layer(fc.style(**{"color": "ff0000", "fillColor": "00000000"}), {},
"Countries")
m

```

23.10.2. From Shapefile

Download and load a shapefile of country boundaries. The shapefile is converted to a FeatureCollection for visualization on the map.

```

url = "https://github.com/gee-community/geemap/blob/master/examples/data/
countries.zip"
geemap.download_file(url, overwrite=True)

```

```

in_shp = "countries.shp"
fc = geemap.shp_to_ee(in_shp)
m = geemap.Map()
m.add_layer(fc, {}, "Countries")
m

```

23.10.3. From GeoDataFrame

Read a shapefile into a GeoDataFrame using geopandas, then convert the GeoDataFrame to an Earth Engine FeatureCollection for mapping.

```

import geopandas as gpd

gdf = gpd.read_file(in_shp)
fc = geemap.gdf_to_ee(gdf)
m = geemap.Map()
m.add_layer(fc, {}, "Countries")
m

```

23.10.4. To GeoJSON

Filter U.S. state data to select Tennessee and save it as a GeoJSON file, which can be shared or used in other GIS tools.

```

m = geemap.Map()
states = ee.FeatureCollection("TIGER/2018/States")
fc = states.filter(ee.Filter.eq("NAME", "Tennessee"))

```



```
m.add_layer(fc, {}, "Tennessee")
m.center_object(fc, 7)
m
```

```
geemap.ee_to_geojson(fc, filename="Tennessee.geojson")
```

23.10.5. To Shapefile

Export the filtered Tennessee FeatureCollection to a shapefile format for offline use or compatibility with desktop GIS software.

```
geemap.ee_to_shp(fc, filename="Tennessee.shp")
```

23.10.6. To GeoDataFrame

Convert an Earth Engine FeatureCollection to a GeoDataFrame for further analysis in Python or use in interactive maps.

```
gdf = geemap.ee_to_gdf(fc)
gdf
```

```
gdf.explore()
```

23.10.7. To DataFrame

Transform an Earth Engine FeatureCollection into a pandas DataFrame, which can then be used for data analysis in Python.

```
df = geemap.ee_to_df(fc)
df
```

23.10.8. To CSV

Export the FeatureCollection data to a CSV file, useful for spreadsheet applications and data reporting.

```
geemap.ee_to_csv(fc, filename="Indiana.csv")
```

23.11. Raster Data Processing

23.11.1. Extract Pixel Values

23.11.1.1. Extracting Values to Points

Load and visualize SRTM DEM and Landsat 7 imagery. Points from a shapefile of U.S. cities are added, and the tool extracts DEM values to these points, saving the results in a shapefile.

```
m = geemap.Map(center=[40, -100], zoom=4)

dem = ee.Image("USGS/SRTMGL1_003")
landsat7 = ee.Image("LANDSAT/LE7_TOA_5YEAR/1999_2003")

vis_params = {
    "min": 0,
    "max": 4000,
    "palette": ["006633", "E5FFCC", "662A00", "D8D8D8", "F5F5F5"],
}

m.add_layer(
    landsat7,
    {"bands": ["B4", "B3", "B2"], "min": 20, "max": 200, "gamma": 2},
    "Landsat 7",
)
m.add_layer(dem, vis_params, "SRTM DEM", True, 1)
m
```

Let's download a shapefile of the world cities from GitHub and convert it to a feature collection.

```
in_shp = "us_cities.shp"
url = "https://github.com/opengeos/data/raw/main/us/us_cities.zip"
geemap.download_file(url)
```

```
in_fc = geemap.shp_to_ee(in_shp)
m.add_layer(in_fc, {}, "Cities")
```

With the DEM and cities, we can now extract the elevation values for each city:

```
geemap.extract_values_to_points(in_fc, dem, out_fc="dem.shp", scale=90)
```

Convert the shapefile to a GeoDataFrame and display the first few rows:

```
gdf = geemap.shp_to_gdf("dem.shp")
gdf.head()
```

In addition to extracting pixel values from a single-band image, we can also extract pixel values from a multi-band image. The example below shows how to extract pixel values from a Landsat image using US cities as points.

```
geemap.extract_values_to_points(in_fc, landsat7, "landsat.csv", scale=90)
```

Convert the CSV file to a pandas DataFrame and display the first few rows.

```
df = geemap.csv_to_df("landsat.csv")
df.head()
```

23.11.1.2. Extracting Pixel Values Along a Transect

In this example, we will extract pixel values along a transect line. First, we will create a line feature collection from a shapefile. Then, we will extract pixel values along the line. Finally, we will plot the pixel values. Let's start by adding the SRTM DEM to the map.

```
m = geemap.Map(center=[40, -100], zoom=4)
m.add_basemap("TERRAIN")

image = ee.Image("USGS/SRTMGL1_003")
vis_params = {
    "min": 0,
    "max": 4000,
    "palette": ["006633", "E5FFCC", "662A00", "D8D8D8", "F5F5F5"],
}
m.add_layer(image, vis_params, "SRTM DEM", True, 0.5)
m
```

Use the drawing tool to draw a line on the map. If no line is drawn, the code below will draw a line from (-120.2232, 36.3148) to (-118.9269, 36.7121) to (-117.2022, 36.7562). Then, run the code below to extract pixel values along the line.

```
line = m.user_roi
if line is None:
    line = ee.Geometry.LineString(
        [[-120.2232, 36.3148], [-118.9269, 36.7121], [-117.2022, 36.7562]]
    )
    m.add_layer(line, {}, "ROI")
m.centerObject(line)
```

The `extract_transect` function can be used to extract pixel values along a line. The function takes an image, a line, and a reducer function. The reducer function is used to reduce the pixel values along the line. The function returns a pandas dataframe with the pixel values and the coordinates of the line.

```

reducer = "mean"
transect = geemap.extract_transect(
    image, line, n_segments=100, reducer=reducer, to_pandas=True
)
transect

```

With the resulting data frame from above, we can plot the elevation profile ([Figure 94](#)).

```

geemap.line_chart(
    data=transect,
    x="distance",
    y="mean",
    markers=True,
    x_label="Distance (m)",
    y_label="Elevation (m)",
    height=400,
)

```

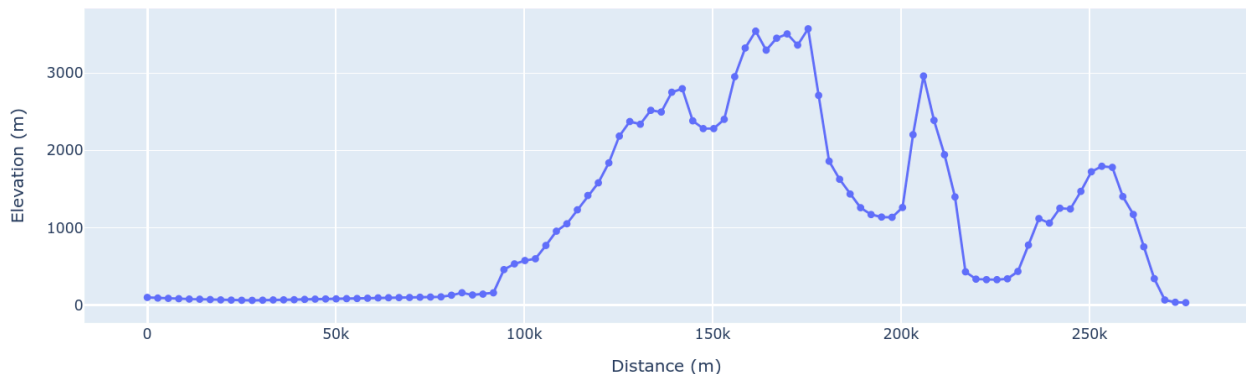


Figure 94: An elevation profile extracted from a DEM.

```

transect.to_csv("transect.csv")

```

23.11.2. Zonal Statistics

23.11.2.1. Zonal Statistics with an Image and a Feature Collection

This section demonstrates the use of zonal statistics to calculate the mean elevation values within U.S. state boundaries using NASA's SRTM DEM data and a 5-year Landsat composite. The `zonal_stats()` function exports results to CSV files for analysis.

First, let's load the SRTM DEM and the Landsat composite:

```

m = geemap.Map(center=[40, -100], zoom=4)

```

```

# Add NASA SRTM
dem = ee.Image("USGS/SRTMGL1_003")
dem_vis = {
    "min": 0,
    "max": 4000,
    "palette": ["006633", "E5FFCC", "662A00", "D8D8D8", "F5F5F5"],
}
m.add_layer(dem, dem_vis, "SRTM DEM")

# Add 5-year Landsat TOA composite
landsat = ee.Image("LANDSAT/LE7_TOA_5YEAR/1999_2003")
landsat_vis = {"bands": ["B4", "B3", "B2"], "gamma": 1.4}
m.add_layer(landsat, landsat_vis, "Landsat", False)

# Add US Census States
states = ee.FeatureCollection("TIGER/2018/States")
style = {"fillColor": "00000000"}
m.add_layer(states.style(**style), {}, "US States")
m

```

Run the code below to calculate the mean elevation values within U.S. state boundaries using NASA's SRTM DEM data:

```

out_dem_stats = "dem_stats.csv"
geemap.zonal_stats(
    dem, states, out_dem_stats, statistics_type="MEAN", scale=1000,
    return_fc=False
)

```

Run the code below to calculate the mean spectral values within U.S. state boundaries using the Landsat composite and the U.S. state boundaries:

```

out_landsat_stats = "landsat_stats.csv"
geemap.zonal_stats(
    landsat,
    states,
    out_landsat_stats,
    statistics_type="MEAN",
    scale=1000,
    return_fc=False,
)

```

23.11.2.2. Zonal Statistics by Group

Here, zonal statistics are applied to NLCD land cover data, calculating the area of each land cover type within each U.S. state. The results are saved to CSV files as both raw totals and percentages. This provides insights into the spatial distribution of land cover categories across states.

```

m = geemap.Map(center=[40, -100], zoom=4)

# Add NLCD data
dataset = ee.Image("USGS/NLCD_RELEASES/2019_REL/NLCD/2019")
landcover = dataset.select("landcover")
m.add_layer(landcover, {}, "NLCD 2019")

# Add US census states
states = ee.FeatureCollection("TIGER/2018/States")
style = {"fillColor": "00000000"}
m.add_layer(states.style(**style), {}, "US States")

# Add NLCD legend
m.add_legend(title="NLCD Land Cover", builtin_legend="NLCD")
m

```

```

nlcd_stats = "nlcd_stats.csv"

geemap.zonal_stats_by_group(
    landcover,
    states,
    nlcd_stats,
    stat_type="SUM",
    denominator=1e6,
    decimal_places=2,
    scale=300
)

```

```

nlcd_stats = "nlcd_stats_pct.csv"

geemap.zonal_stats_by_group(
    landcover,
    states,
    nlcd_stats,
    stat_type="PERCENTAGE",
    denominator=1e6,
    decimal_places=2,
    scale=300
)

```

23.11.2.3. Zonal Statistics with Two Images

This example calculates the mean elevation values within different NLCD land cover types using DEM and NLCD data. The `image_stats_by_zone()` function provides summary statistics (e.g., mean and standard deviation), which can be exported to CSV files for further analysis or visualization.

```

m = geemap.Map(center=[40, -100], zoom=4)
dem = ee.Image("USGS/3DEP/10m")
vis = {"min": 0, "max": 4000, "palette": "terrain"}
m.add_layer(dem, vis, "DEM")
landcover = ee.Image("USGS/NLCD_RELEASES/2019_REL/NLCD/2019").select("landcover")
m.add_layer(landcover, {}, "NLCD 2019")
m.add_legend(title="NLCD Land Cover Classification", builtin_legend="NLCD")
m

```

```

stats = geemap.image_stats_by_zone(dem, landcover, reducer="MEAN", scale=90)
stats

```

```

stats.to_csv("mean.csv", index=False)

```

```

geemap.image_stats_by_zone(dem, landcover, out_csv="std.csv", reducer="STD")

```

23.11.3. Map Algebra

This example demonstrates basic map algebra by computing the Normalized Difference Vegetation Index (NDVI) for a 5-year Landsat composite and the Enhanced Vegetation Index (EVI) for a Landsat 8 image. These indices are visualized with color scales to highlight areas of vegetation ([Figure 95](#)).

```

m = geemap.Map()

# Load a 5-year Landsat 7 composite 1999-2003.
landsat_1999 = ee.Image("LANDSAT/LE7_TOA_5YEAR/1999_2003")

# Compute NDVI.
ndvi_1999 = (
    landsat_1999.select("B4")
    .subtract(landsat_1999.select("B3"))
    .divide(landsat_1999.select("B4").add(landsat_1999.select("B3")))
)

vis = {"min": 0, "max": 1, "palette": "ndvi"}
m.add_layer(ndvi_1999, vis, "NDVI")
m.add_colorbar(vis, label="NDVI")
m

```

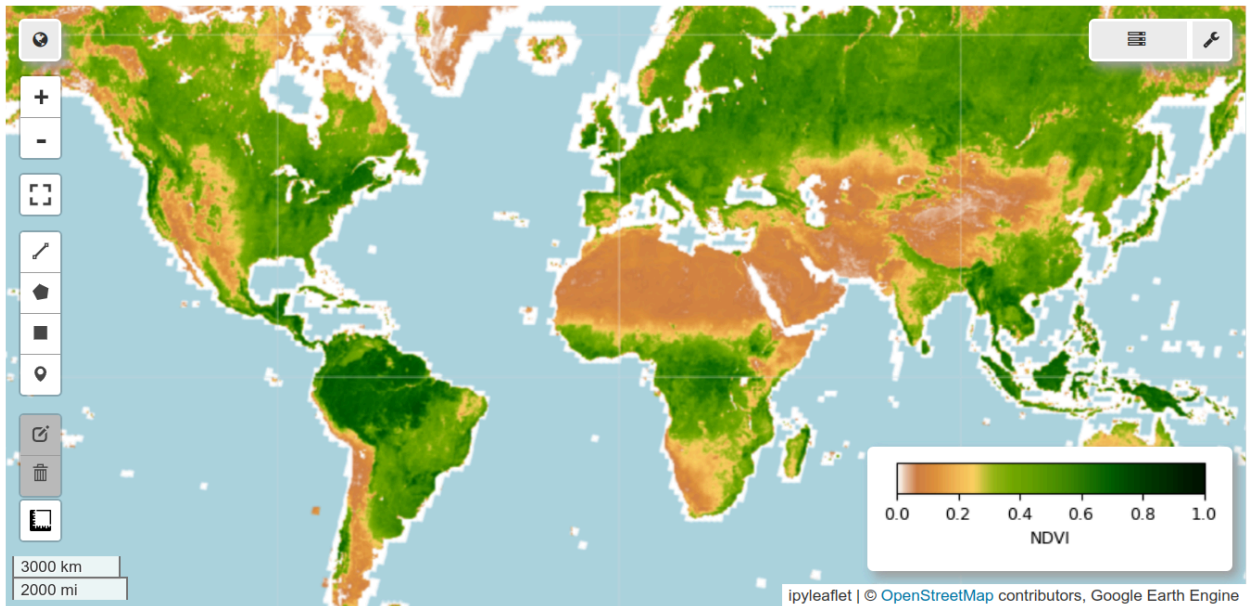


Figure 95: NDVI visualization of Landsat 7 data.

```
# Load a Landsat 8 image.
image = ee.Image("LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318")

# Compute the EVI using an expression.
evi = image.expression(
    "2.5 * ((NIR - RED) / (NIR + 6 * RED - 7.5 * BLUE + 1))",
    {
        "NIR": image.select("B5"),
        "RED": image.select("B4"),
        "BLUE": image.select("B2"),
    },
)

# Define a map centered on San Francisco Bay.
m = geemap.Map(center=[37.4675, -122.1363], zoom=9)

vis = {"min": 0, "max": 1, "palette": "ndvi"}
m.add_layer(evi, vis, "EVI")
m.add_colorbar(vis, label="EVI")
m
```

23.12. Exporting Earth Engine Data

23.12.1. Exporting Images

This section demonstrates exporting Earth Engine images. First, a Landsat image is added to the map for visualization, and a rectangular region of interest (ROI) is specified. Exporting options include saving the

image locally with specified scale and region or exporting to Google Drive. It's possible to define custom CRS and transformation settings when saving the image.

Add a Landsat image to the map.

```
m = geemap.Map()

image = ee.Image("LANDSAT/LC08/C02/T1_TOA/LC08_044034_20140318").select(
    ["B5", "B4", "B3"]
)

vis_params = {"min": 0, "max": 0.5, "gamma": [0.95, 1.1, 1]}

m.center_object(image)
m.add_layer(image, vis_params, "Landsat")
m
```

Add a rectangle to the map.

```
region = ee.Geometry.BBox(-122.5955, 37.5339, -122.0982, 37.8252)
fc = ee.FeatureCollection(region)
style = {"color": "ffff00ff", "fillColor": "00000000"}
m.add_layer(fc.style(**style), {}, "ROI")
```

Export the NDVI image to a GeoTIFF file.

```
geemap.ee_export_image(image, filename="landsat.tif", scale=30, region=region)
```

Check the image projection.

```
projection = image.select(0).projection().getInfo()
projection
```

```
crs = projection["crs"]
crs_transform = projection["transform"]
```

If you need to export multiple images covering the same region, you can specify the region, crs, and crs_transform so that the exported images can align with each other.

```
geemap.ee_export_image(
    image,
    filename="landsat_crs.tif",
    crs=crs,
    crs_transform=crs_transform,
    region=region,
)
```

The `ee_export_image` function introduced above can only export a small image (< 32 MB). If you need to export a larger image, you can use the `ee_export_image_to_drive` function to export the image to Google Drive, which can handle much larger images.

```
geemap.ee_export_image_to_drive(  
    image, description="landsat", folder="export", region=region, scale=30  
)
```

Alternatively, you can use the `download_ee_image()` function to download the image to your local machine.

```
geemap.download_ee_image(image, "landsat.tif", scale=90)
```

23.12.2. Exporting Image Collections

Image collections, like time series data, can be filtered and exported as multiple images. Here, a National Agriculture Imagery Program (NAIP) collection is filtered by date and location. The collection can then be saved locally or sent to Google Drive, making it easier to handle large datasets.

```
point = ee.Geometry.Point(-99.2222, 46.7816)  
collection = (  
    ee.ImageCollection("USDA/NAIP/DOQQ")  
    .filterBounds(point)  
    .filterDate("2008-01-01", "2018-01-01")  
    .filter(ee.Filter.listContains("system:band_names", "N"))  
)  
collection.aggregate_array("system:index")
```

First, let's export all images in the collection to a local folder.

```
geemap.ee_export_image_collection(collection, out_dir=".", scale=10)
```

Alternatively, you can use export all images in the collection to Google Drive.

```
geemap.ee_export_image_collection_to_drive(collection, folder="export", scale=10)
```

23.12.3. Exporting Feature Collections

Feature collections, such as state boundaries, are exportable in multiple formats (e.g., Shapefile, GeoJSON, and CSV). This example exports the Alaska state boundary as different vector formats both locally and to Google Drive. Additionally, exported data can be directly loaded into a GeoDataFrame for further manipulation in Python.

First, let's load the Alaska state boundary:

```

m = geemap.Map()
states = ee.FeatureCollection("TIGER/2018/States")
fc = states.filter(ee.Filter.eq("NAME", "Alaska"))
m.add_layer(fc, {}, "Alaska")
m.center_object(fc, 4)
m

```

Let's export the Alaska state boundary to various formats, including Shapefile, GeoJSON, CSV, GeoPandas GeoDataFrame, and Pandas Dataframe.

```

geemap.ee_to_shp(fc, filename="Alaska.shp")
geemap.ee_export_vector(fc, filename="Alaska.shp")
geemap.ee_to_geojson(fc, filename="Alaska.geojson")
geemap.ee_to_csv(fc, filename="Alaska.csv")
gdf = geemap.ee_to_gdf(fc)
df = geemap.ee_to_df(fc)

```

Export the feature collection to a shapefile in Google Drive.

```

geemap.ee_export_vector_to_drive(
    fc, description="Alaska", fileFormat="SHP", folder="export"
)

```

23.13. Creating Timelapse Animations

Timelapse animations are one of the most powerful visualization tools for understanding temporal changes in the Earth's surface. These animations compress years or decades of satellite observations into seconds of video, making long-term environmental changes visible and comprehensible. Google Earth Engine's vast temporal archives combined with geemap's timelapse functionality enable the creation of compelling visualizations that reveal patterns of urban growth, deforestation, glacier retreat, seasonal vegetation cycles, and natural disasters.

Geemap provides several specialized functions for creating timelapses from different Earth Engine datasets, each optimized for specific satellite sensors and use cases. The choice of satellite data depends on your temporal requirements, spatial resolution needs, and the phenomena you want to visualize. Let's explore the main timelapse creation tools available in geemap.

23.13.1. Landsat Timelapse

Landsat satellites provide the longest continuous record of Earth's land surface, with data spanning from 1972 to the present. This makes Landsat ideal for visualizing long-term land cover changes, urban expansion, and environmental transformations. The `landsat_timelapse` function automatically handles the complexity of Landsat data processing, including cloud masking, atmospheric correction, and temporal compositing.

The function works by selecting the best available Landsat images for each time period, creating annual or seasonal composites, and assembling them into an animated GIF. You can customize the band combi-

nations to highlight different features: natural color (Red, Green, Blue) for intuitive visualization, false color (NIR, Red, Green) for vegetation analysis, or other combinations for specific applications.

Here, we create a timelapse animation showing the dramatic urban expansion of Las Vegas from 1984 to 2025:

```
m = geemap.Map()
roi = ee.Geometry.BBox(-115.5541, 35.8044, -113.9035, 36.5581)
m.add_layer(roi)
m.center_object(roi)
m
```

```
timelapse = geemap.landsat_timelapse(
    roi,
    out_gif="las_vegas.gif",
    start_year=1984,
    end_year=2025,
    bands=["NIR", "Red", "Green"],
    frames_per_second=5,
    title="Las Vegas, NV",
    font_color="blue",
)
geemap.show_image(timelapse)
```

The Las Vegas example uses a false-color band combination (NIR, Red, Green) that makes urban areas appear blue-gray while vegetation appears red. This combination is particularly effective for visualizing urban development and land use changes.

Let's create another example showing rapid development in Hong Kong:

```
m = geemap.Map()
roi = ee.Geometry.BBox(113.8252, 22.1988, 114.0851, 22.3497)
m.add_layer(roi)
m.center_object(roi)
m
```

```
timelapse = geemap.landsat_timelapse(
    roi,
    out_gif="hong_kong.gif",
    start_year=1990,
    end_year=2025,
    start_date="01-01",
    end_date="12-31",
    bands=["SWIR1", "NIR", "Red"],
    frames_per_second=3,
    title="Hong Kong",
)
```

```
)
geemap.show_image(timelapse)
```

This Hong Kong example demonstrates land reclamation and urban development over three decades. The SWIR1, NIR, Red band combination effectively highlights the contrast between water, vegetation, and built environments.

23.13.2. Sentinel-2 Timelapse

Sentinel-2 satellites, launched by the European Space Agency, provide high-resolution (10-20 meter) multispectral imagery with a revisit time of approximately 5 days. While Sentinel-2's temporal record is shorter than Landsat (beginning in 2015), it offers superior spatial resolution and more frequent observations, making it ideal for monitoring rapid changes or seasonal phenomena that require fine spatial detail.

The `sentinel2_timelapse` function is particularly useful for agricultural monitoring, seasonal vegetation analysis, and tracking short-term environmental changes. The high temporal frequency allows for smooth animations that capture gradual transitions and seasonal cycles.

This example generates a timelapse showing river dynamics in South America.

```
m = geemap.Map(center=[-8.4506, -74.5079], zoom=12)
m
```

Pan and zoom the map to an area of interest. Use the drawing tools to draw a rectangle on the map. If no rectangle is drawn, the default rectangle shown below will be used.

```
roi = m.user_roi
if roi is None:
    roi = ee.Geometry.BBox(-74.7222, -8.5867, -74.1596, -8.2824)
    m.add_layer(roi)
    m.center_object(roi)
```

```
timelapse = geemap.sentinel2_timelapse(
    roi,
    out_gif="sentinel2.gif",
    start_year=2017,
    end_year=2025,
    start_date="05-01",
    end_date="09-31",
    frequency="year",
    bands=["SWIR1", "NIR", "Red"],
    frames_per_second=3,
    title="Sentinel-2 Timelapse",
)
geemap.show_image(timelapse)
```

The Sentinel-2 animation uses the SWIR1, NIR, Red band combination, which is excellent for distinguishing between water, vegetation, and built environments.

23.13.3. MODIS Timelapse

The Moderate Resolution Imaging Spectroradiometer (MODIS) provides a unique perspective on Earth system processes through its global coverage and daily observations. While MODIS has coarser spatial resolution (250m-1km) compared to Landsat or Sentinel-2, its high temporal frequency and global extent make it ideal for monitoring large-scale environmental phenomena such as vegetation phenology, sea surface temperatures, and atmospheric conditions.

MODIS timelapses are particularly valuable for understanding continental and global-scale processes. The `modis_ndvi_timelapse` function specializes in vegetation monitoring using the Normalized Difference Vegetation Index (NDVI), which indicates vegetation health and photosynthetic activity. The `modis_ocean_color_timelapse` function focuses on marine environments, tracking parameters like sea surface temperature and chlorophyll concentrations.

Let's create a continental-scale vegetation timelapse showing seasonal cycles across Africa and Europe:

```
m = geemap.Map()
m
```

```
roi = m.user_roi
if roi is None:
    roi = ee.Geometry.BBox(-18.6983, -36.1630, 52.2293, 38.1446)
m.add_layer(roi)
m.center_object(roi)
```

```
timelapse = geemap.modis_ndvi_timelapse(
    roi,
    out_gif="ndvi.gif",
    data="Terra",
    band="NDVI",
    start_date="2000-01-01",
    end_date="2022-12-31",
    frames_per_second=3,
    title="MODIS NDVI Timelapse",
    overlay_data="countries",
)
geemap.show_image(timelapse)
```

This NDVI timelapse reveals the “green wave” phenomenon as vegetation responds to seasonal temperature and precipitation patterns. The country overlays provide geographic context, helping viewers understand how vegetation cycles vary across different climate zones and latitudes.

Now let's create a global ocean temperature timelapse to demonstrate marine monitoring capabilities:

```
m = geemap.Map()
m
```

```
roi = m.user_roi
if roi is None:
    roi = ee.Geometry.BBox(-171.21, -57.13, 177.53, 79.99)
    m.add_layer(roi)
    m.center_object(roi)
```

```
timelapse = geemap.modis_ocean_color_timelapse(
    satellite="Aqua",
    start_date="2018-01-01",
    end_date="2020-12-31",
    roi=roi,
    frequency="month",
    out_gif="temperature.gif",
    overlay_data="continents",
    overlay_color="yellow",
    overlay_opacity=0.5,
)
geemap.show_image(timelapse)
```

This ocean temperature animation demonstrates the seasonal warming and cooling of Earth's oceans, revealing patterns like El Niño/La Niña cycles and the formation of seasonal temperature gradients. The continental overlays help viewers understand the relationship between land masses and ocean temperature patterns.

23.13.4. GOES Timelapse

The Geostationary Operational Environmental Satellites (GOES) provide a unique perspective on Earth's atmosphere from a fixed position above the equator. Unlike polar-orbiting satellites that pass over the same location once or twice daily, GOES satellites can observe the same region continuously, making them ideal for tracking fast-moving atmospheric phenomena such as hurricanes, thunderstorms, and wildfires.

GOES timelapses excel at capturing the dynamics of weather systems and natural disasters, with temporal resolution measured in minutes rather than days. The `goes_timelapse` and `goes_fire_timelapse` functions can create dramatic animations showing the evolution of atmospheric events in near real-time.

Let's start with a basic GOES animation showing the Tonga volcanic eruption:

```
roi = ee.Geometry.BBox(167.1898, -28.5757, 202.6258, -12.4411)
start_date = "2022-01-15T03:00:00"
end_date = "2022-01-15T07:00:00"
data = "GOES-17"
scan = "full_disk"
```

```
timelapse = geemap.goes_timelapse(
    roi, "goes.gif", start_date, end_date, data, scan, framesPerSecond=5
)
geemap.show_image(timelapse)
```

Now let's create a hurricane tracking animation:

```
roi = ee.Geometry.BBox(-159.5954, 24.5178, -114.2438, 60.4088)
start_date = "2021-10-24T14:00:00"
end_date = "2021-10-25T01:00:00"
data = "GOES-17"
scan = "full_disk"
```

```
timelapse = geemap.goes_timelapse(
    roi, "hurricane.gif", start_date, end_date, data, scan, framesPerSecond=5
)
geemap.show_image(timelapse)
```

This hurricane animation demonstrates the dramatic cloud formations and eye wall structure of a major storm system, showing how GOES data can be used for weather forecasting and disaster monitoring.

Finally, let's create a wildfire monitoring animation using the specialized fire detection algorithm:

```
roi = ee.Geometry.BBox(-121.0034, 36.8488, -117.9052, 39.0490)
start_date = "2020-09-05T15:00:00"
end_date = "2020-09-06T02:00:00"
data = "GOES-17"
scan = "full_disk"
```

```
timelapse = geemap.goes_fire_timelapse(
    roi, "fire.gif", start_date, end_date, data, scan, framesPerSecond=5
)
geemap.show_image(timelapse)
```

The fire timelapse uses specialized algorithms to highlight thermal anomalies associated with wildfires, providing critical information for fire management and emergency response efforts.

23.14. Charting Earth Engine Data

Data visualization extends beyond interactive maps to include statistical charts and graphs that reveal patterns, trends, and relationships in geospatial datasets. While maps excel at showing spatial patterns, charts are essential for communicating temporal trends, statistical distributions, and quantitative relationships between variables. Earth Engine's massive computational power combined with geemap's charting capabilities enables the creation of publication-quality visualizations directly from satellite data and other geospatial sources.

The geemap charting system provides a comprehensive toolkit for creating various types of charts from Earth Engine data, including time series plots, histograms, scatter plots, and statistical summaries. These charts can be generated from features, images, image collections, and computed statistics, making it possible to visualize everything from individual pixel values to continental-scale phenomena.

Understanding when and how to use different chart types is crucial for effective geospatial data analysis. Time series charts reveal temporal patterns and trends, histograms show data distributions, scatter plots reveal relationships between variables, and bar charts enable comparisons between categories or regions. The following sections demonstrate these fundamental charting techniques using real Earth Engine datasets.

23.14.1. Charting Features

Feature-based charts are particularly useful for analyzing and comparing attributes across different geographic regions or administrative boundaries. These charts can display properties stored in Earth Engine FeatureCollections, such as census data, environmental measurements, or calculated statistics for different areas. The charting functions transform tabular attribute data into visual representations that make patterns and relationships more apparent.

23.14.1.1. Import libraries

To create charts for Earth Engine features, import the geemap's chart module.

```
import calendar
from geemap import chart
```

23.14.1.2. feature_by_feature

The `feature_by_feature` function creates charts where individual features (such as geographic regions) are plotted along the x-axis, with multiple properties displayed as different data series. This chart type is ideal for comparing multiple attributes across different geographic areas, such as comparing monthly temperature or precipitation values across different ecoregions.

In this example, we visualize average monthly temperature data across different ecoregions, with each month represented as a separate data series (Figure 96).

```
ecoregions = ee.FeatureCollection("projects/google/charts_feature_example")
features = ecoregions.select("[0-9][0-9]_tmean|label")
geemap.ee_to_df(features)
```

```
x_property = "label"
y_properties = [str(x).zfill(2) + "_tmean" for x in range(1, 13)]

labels = calendar.month_abbr[1:] # a list of month labels, e.g. ['Jan',
'Feb', ...]

colors = [
    "#604791",
```

```

"#1d6b99",
"#39a8a7",
"#0f8755",
"#76b349",
"#f0af07",
"#e37d05",
"#cf513e",
"#96356f",
"#724173",
"#9c4f97",
"#696969",
]
title = "Average Monthly Temperature by Ecoregion"
x_label = "Ecoregion"
y_label = "Temperature"

fig = chart.feature_by_feature(
    features,
    x_property,
    y_properties,
    colors=colors,
    labels=labels,
    title=title,
    x_label=x_label,
    y_label=y_label,
)
fig

```

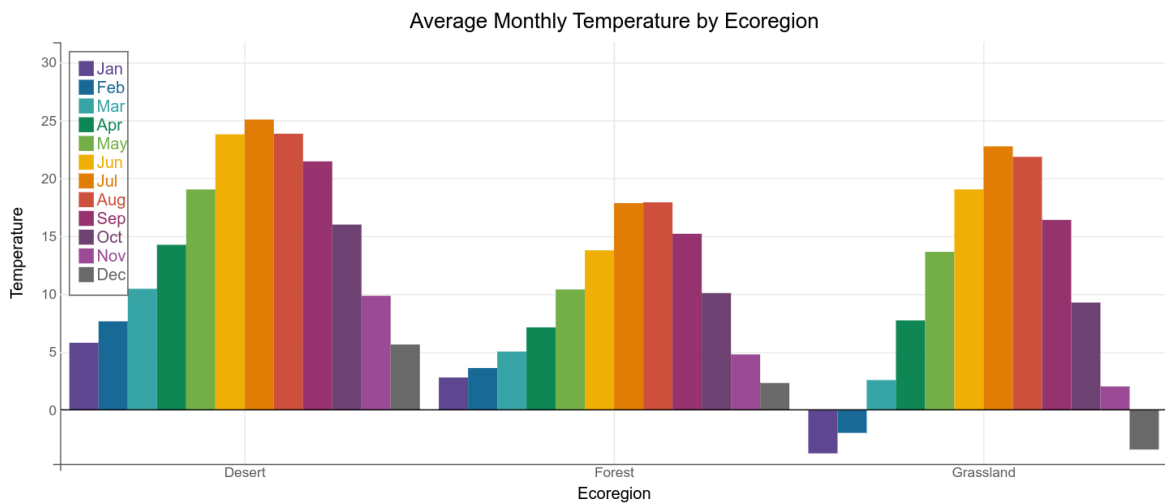


Figure 96: A chart showing the average monthly temperature by ecoregion.

23.14.1.3. feature.by_property

The `feature_by_property` function creates charts where properties (such as months or categories) are plotted along the x-axis, with different features displayed as separate data series. This chart type is particularly useful for examining temporal patterns or categorical comparisons, showing how different regions respond to seasonal or categorical variations.

This example displays average precipitation by month for different ecoregions, allowing us to compare seasonal precipitation patterns across different ecological zones ([Figure 97](#)):

```
ecoregions = ee.FeatureCollection("projects/google/charts_feature_example")
features = ecoregions.select("[0-9][0-9]_ppt|label")
geemap.ee_to_df(features)
```

```
keys = [str(x).zfill(2) + "_ppt" for x in range(1, 13)]
values = calendar.month_abbr[1:] # a list of month labels, e.g. ['Jan',
'Feb', ...]
x_properties = dict(zip(keys, values))
series_property = "label"
title = "Average Ecoregion Precipitation by Month"
colors = ["#f0af07", "#0f8755", "#76b349"]
fig = chart.feature_by_property(
    features,
    x_properties,
    series_property,
    title=title,
    colors=colors,
    x_label="Month",
    y_label="Precipitation (mm)",
    legend_location="top-left",
)
fig
```

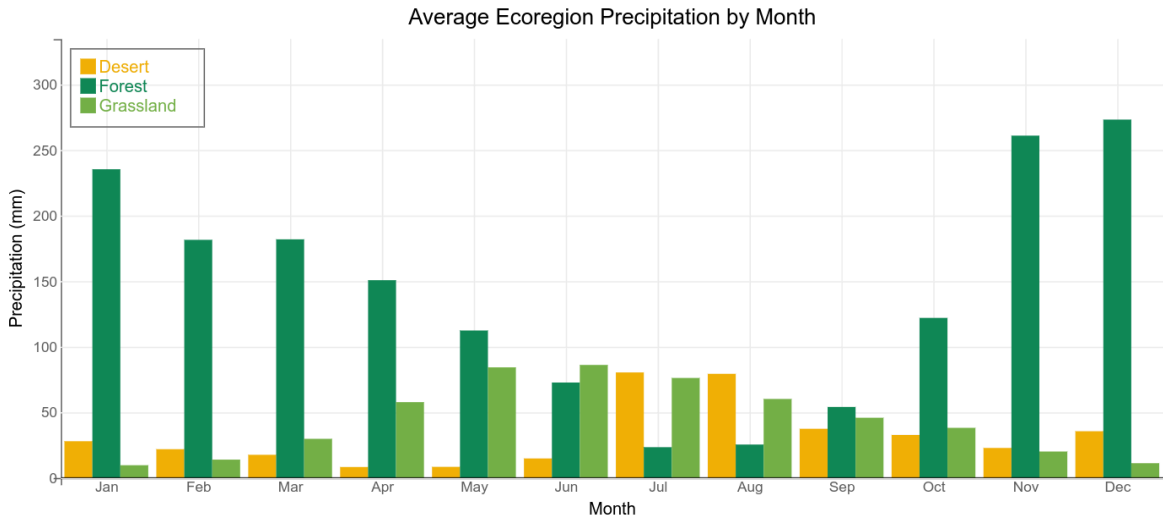


Figure 97: A chart showing the average monthly precipitation by ecoregion.

23.14.1.4. feature_groups

The `feature_groups` function creates charts that group features based on categorical properties, allowing for comparison between different categories or classes. This is particularly useful for analyzing how different groups of features differ in their measured properties, such as comparing environmental conditions between different land cover types or climate zones.

In this example, we compare average January temperatures between ecoregions classified as “Warm” and “Cold”, demonstrating how categorical grouping can reveal climate-based patterns (Figure 98):

```
ecoregions = ee.FeatureCollection("projects/google/charts_feature_example")
features = ecoregions.select("[0-9][0-9]_ppt|label")
features = ee.FeatureCollection("projects/google/charts_feature_example")
x_property = "label"
y_property = "01_tmean"
series_property = "warm"
title = "Average January Temperature by Ecoregion"
colors = ["#cf513e", "#1d6b99"]
labels = ["Warm", "Cold"]

fig = chart.feature_groups(
    features,
    x_property,
    y_property,
    series_property,
    title=title,
    colors=colors,
    x_label="Ecoregion",
    y_label="January Temperature (°C)",
```

```

    legend_location="top-right",
    labels=labels,
)
fig

```

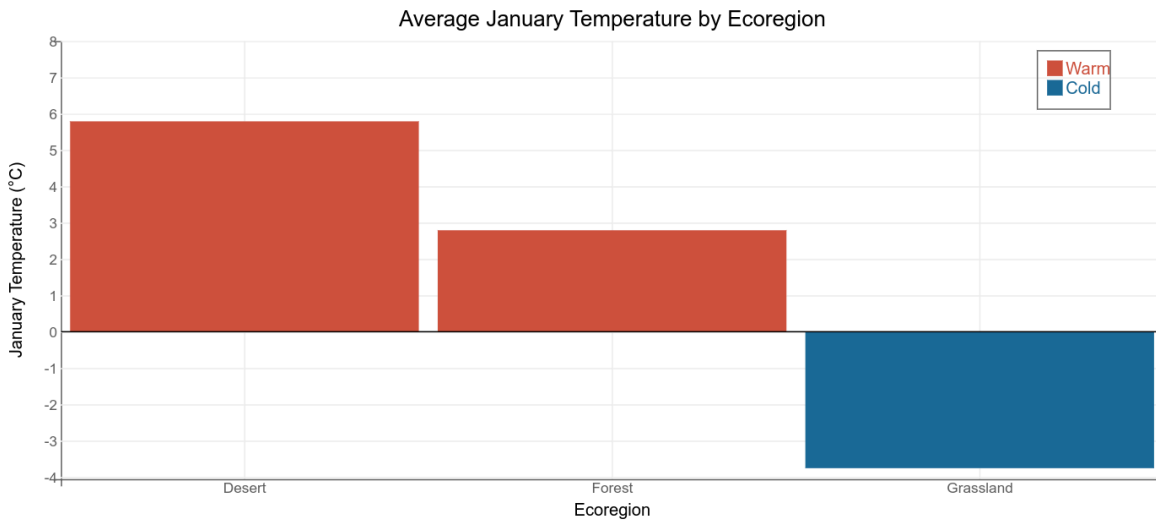


Figure 98: A chart showing the average January temperature by ecoregion.

23.14.1.5. feature_histogram

The `feature_histogram` function creates frequency distribution charts that show how values are distributed across a dataset. Histograms are essential for understanding data characteristics such as central tendency, spread, and the presence of outliers or multiple modes. This chart type is particularly valuable for quality assessment and statistical analysis of geospatial data.

This example generates a histogram showing the distribution of July precipitation values across the northwestern United States, revealing the range and frequency of precipitation levels in this region (Figure 99):

```

source = ee.ImageCollection("OREGONSTATE/PRISM/Norm91m").toBands()
region = ee.Geometry.Rectangle(-123.41, 40.43, -116.38, 45.14)
features = source.sample(region, 5000)
geemap.ee_to_df(features.limit(5).select(["07_ppt"]))

```

```

property = "07_ppt"
title = "July Precipitation Distribution for NW USA"
fig = chart.feature_histogram(
    features,
    property,
    max_buckets=None,

```

```

title=title,
x_label="Precipitation (mm)",
y_label="Pixel Count",
colors=["#1d6b99"],
)
fig

```

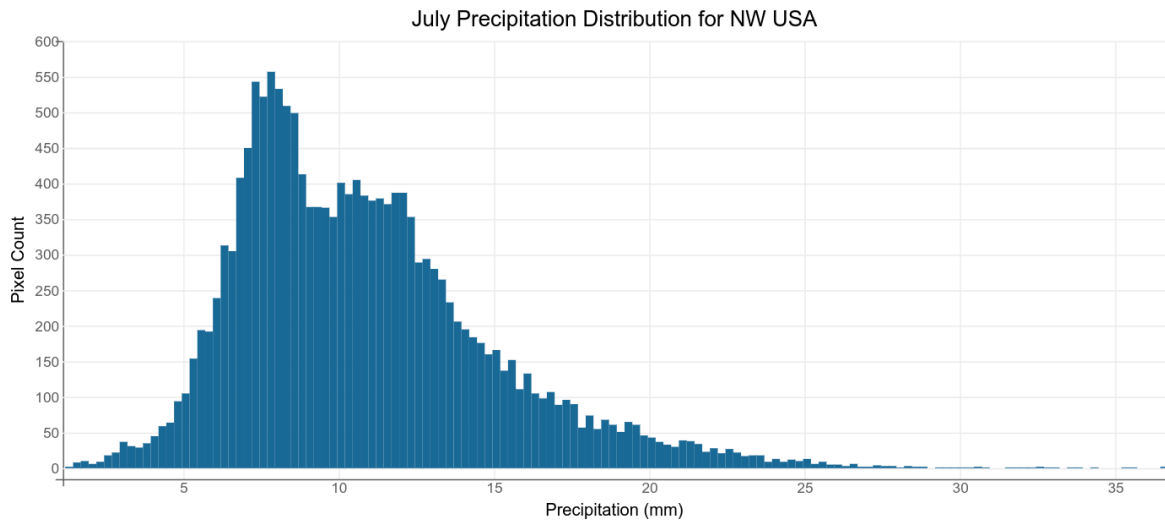
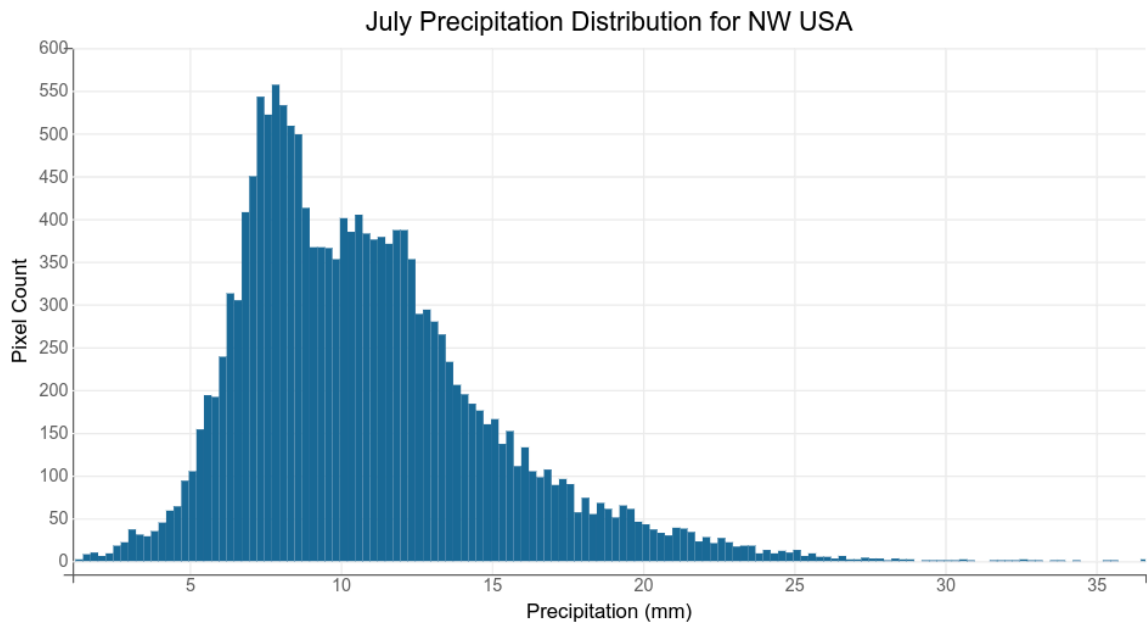


Figure 99: A histogram showing the distribution of July precipitation in the northwest United States.



23.14.2. Charting Images

Image-based charts extract statistical information from Earth Engine Images and present it in graphical form. These charts are particularly useful for analyzing how image values vary across different geographic regions, spectral bands, or statistical measures. Unlike feature-based charts that work with pre-computed attributes, image charts perform spatial aggregation operations to derive statistics from pixel values.

23.14.2.1. image_by_region

The `image_by_region` function computes statistics from an image across different spatial regions, then visualizes these statistics as a chart. This approach is particularly useful for comparing environmental conditions across different administrative boundaries, ecoregions, or other spatial units.

This example extracts monthly temperature data from PRISM climate images and displays average temperature values for different ecoregions ([Figure 100](#)):

```
ecoregions = ee.FeatureCollection("projects/google/charts_feature_example")
image = (
    ee.ImageCollection("OREGONSTATE/PRISM/Norm91m").toBands().select(["0-9]
[0-9]_tmean")
)
labels = calendar.month_abbr[1:] # a list of month labels, e.g. ['Jan',
'Feb', ...]
title = "Average Monthly Temperature by Ecoregion"
fig = chart.image_by_region(
    image,
    ecoregions,
    reducer="mean",
    scale=500,
    x_property="label",
    title=title,
    x_label="Ecoregion",
    y_label="Temperature",
    labels=labels,
)
fig
```

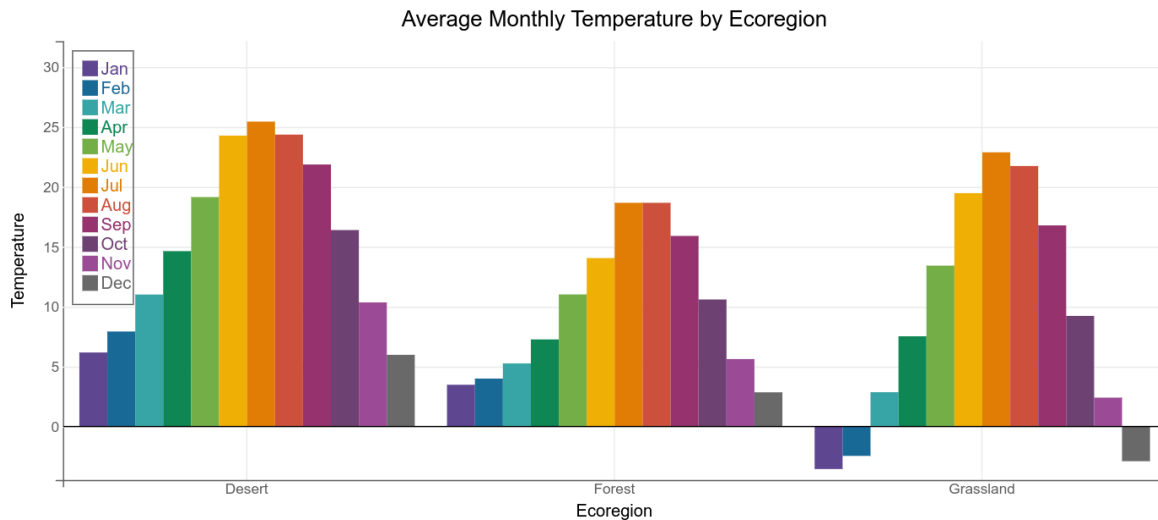


Figure 100: A chart showing the average monthly temperature by ecoregion.

23.14.2.2. image_regions

Generates a chart showing average monthly precipitation across regions, using properties to represent months and comparing precipitation levels (Figure 101):

```
ecoregions = ee.FeatureCollection("projects/google/charts_feature_example")
image = (
    ee.ImageCollection("OREGONSTATE/PRISM/Norm91m").toBands().select("[0-9]
[0-9]_ppt")
)
keys = [str(x).zfill(2) + "_ppt" for x in range(1, 13)]
values = calendar.month_abbr[1:] # a list of month labels, e.g. ['Jan',
'Feb', ...]
x_properties = dict(zip(keys, values))
title = "Average Ecoregion Precipitation by Month"
colors = ["#f0af07", "#0f8755", "#76b349"]
fig = chart.image_regions(
    image,
    ecoregions,
    reducer="mean",
    scale=500,
    series_property="label",
    x_labels=x_properties,
    title=title,
    colors=colors,
    x_label="Month",
    y_label="Precipitation (mm)",
    legend_location="top-left",
```



```
)  
fig
```

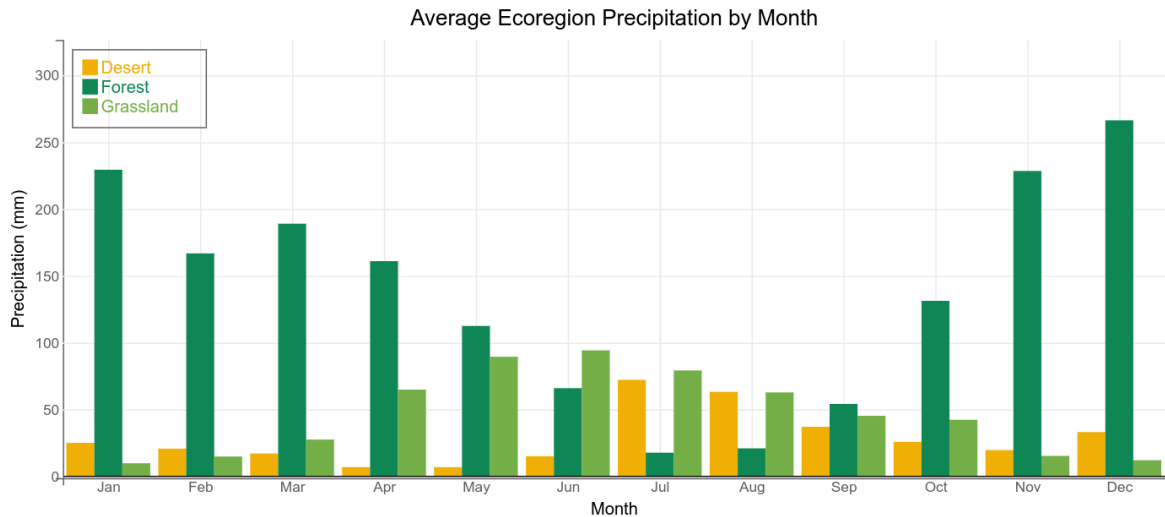


Figure 101: A chart showing the average monthly precipitation by ecoregion.

23.14.2.3. image_by_class

Displays spectral signatures of ecoregions by wavelength, showing reflectance values across spectral bands for comparison (Figure 102):

```
ecoregions = ee.FeatureCollection("projects/google/charts_feature_example")  
image = (  
    ee.ImageCollection("MODIS/061/MOD09A1")  
    .filter(ee.Filter.date("2018-06-01", "2018-09-01"))  
    .select("sur_refl_b0[0-7]")  
    .mean()  
    .select([2, 3, 0, 1, 4, 5, 6])  
)  
wavelengths = [469, 555, 655, 858, 1240, 1640, 2130]  
fig = chart.image_by_class(  
    image,  
    class_band="label",  
    region=ecoregions,  
    reducer="MEAN",  
    scale=500,  
    x_labels=wavelengths,  
    title="Ecoregion Spectral Signatures",  
    x_label="Wavelength (nm)",  
    y_label="Reflectance (x1e4)",
```

```

colors=["#f0af07", "#0f8755", "#76b349"],
legend_location="top-left",
interpolation="basis",
)
fig

```

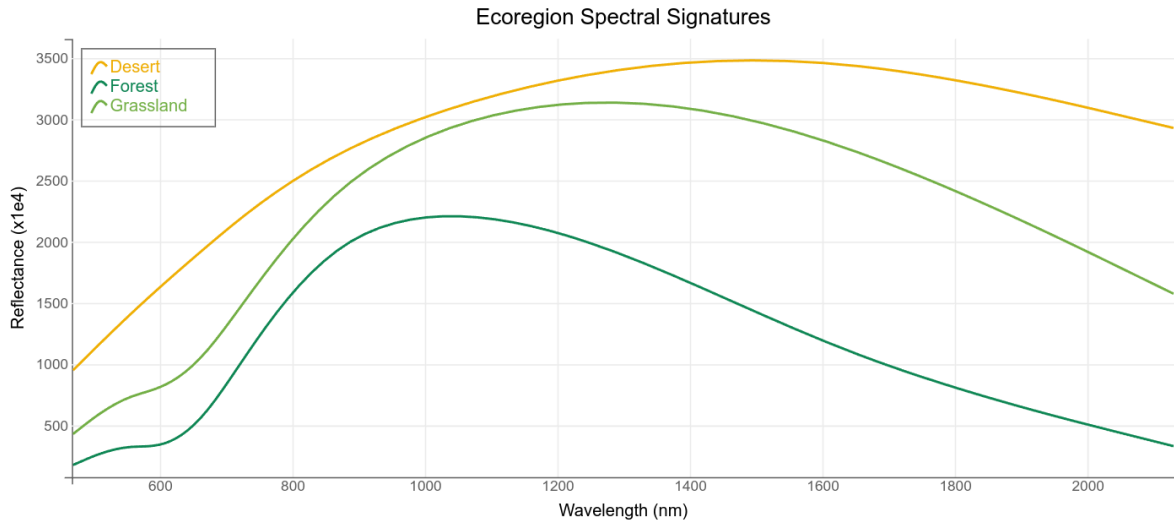


Figure 102: A chart showing the spectral signatures of ecoregions.

23.14.2.4. image_histogram

Generates histograms to visualize MODIS surface reflectance distribution across red, NIR, and SWIR bands, providing insights into data distribution by band (Figure 103):

```

image = (
    ee.ImageCollection("MODIS/061/MOD09A1")
    .filter(ee.Filter.date("2018-06-01", "2018-09-01"))
    .select(["sur_refl_b01", "sur_refl_b02", "sur_refl_b06"])
    .mean()
)
region = ee.Geometry.Rectangle([-112.60, 40.60, -111.18, 41.22])
fig = chart.image_histogram(
    image,
    region,
    scale=500,
    max_buckets=200,
    min_bucket_width=1.0,
    max_raw=1000,
    max_pixels=int(1e6),
    title="MODIS SR Reflectance Histogram",
)

```

```

labels=["Red", "NIR", "SWIR"],
colors=["#cf513e", "#1d6b99", "#f0af07"],
)
fig

```

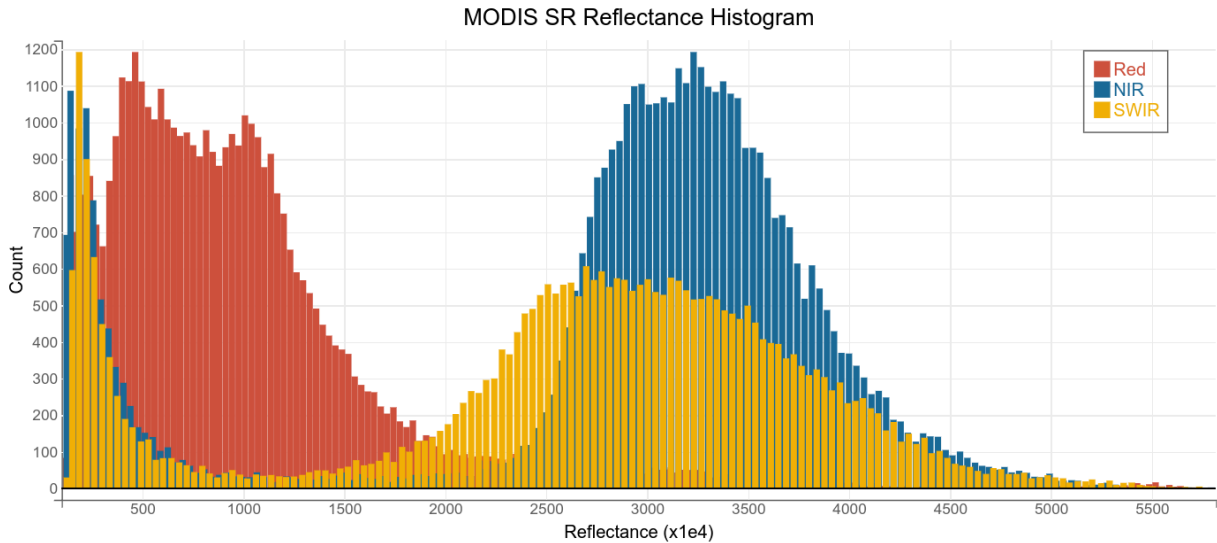


Figure 103: A histogram showing the distribution of MODIS surface reflectance values.

23.14.3. Charting Image Collections

Image collection charts are essential for temporal analysis, allowing you to visualize how environmental conditions change over time. These charts aggregate statistics from multiple images in a collection, creating time series that reveal seasonal patterns, long-term trends, and anomalous events. The temporal dimension adds crucial context that static images cannot provide.

23.14.3.1. image_series

The `image_series` function creates time series charts from Earth Engine ImageCollections, showing how values change over time for a specific location or region. This chart type is fundamental for monitoring environmental change, tracking seasonal cycles, and detecting trends or anomalies in geospatial data.

This example creates a time series showing vegetation health indicators (NDVI and EVI) for a forest region over a decade, revealing seasonal patterns and inter-annual variability ([Figure 104](#)):

```

# Define the forest feature collection.
forest = ee.FeatureCollection("projects/google/charts_feature_example").filter(
    ee.Filter.eq("label", "Forest")
)

# Load MODIS vegetation indices data and subset a decade of images.
veg_indices = (

```

```

ee.ImageCollection("MODIS/061/MOD13A1")
.filter(ee.Filter.date("2010-01-01", "2020-01-01"))
.select(["NDVI", "EVI"])
)

title = "Average Vegetation Index Value by Date for Forest"
x_label = "Year"
y_label = "Vegetation index (x1e4)"
colors = ["#e37d05", "#1d6b99"]

fig = chart.image_series(
    veg_indices,
    region=forest,
    reducer=ee.Reducer.mean(),
    scale=500,
    x_property="system:time_start",
    chart_type="LineChart",
    x_cols="date",
    y_cols=["NDVI", "EVI"],
    colors=colors,
    title=title,
    x_label=x_label,
    y_label=y_label,
    legend_location="right",
)
fig

```

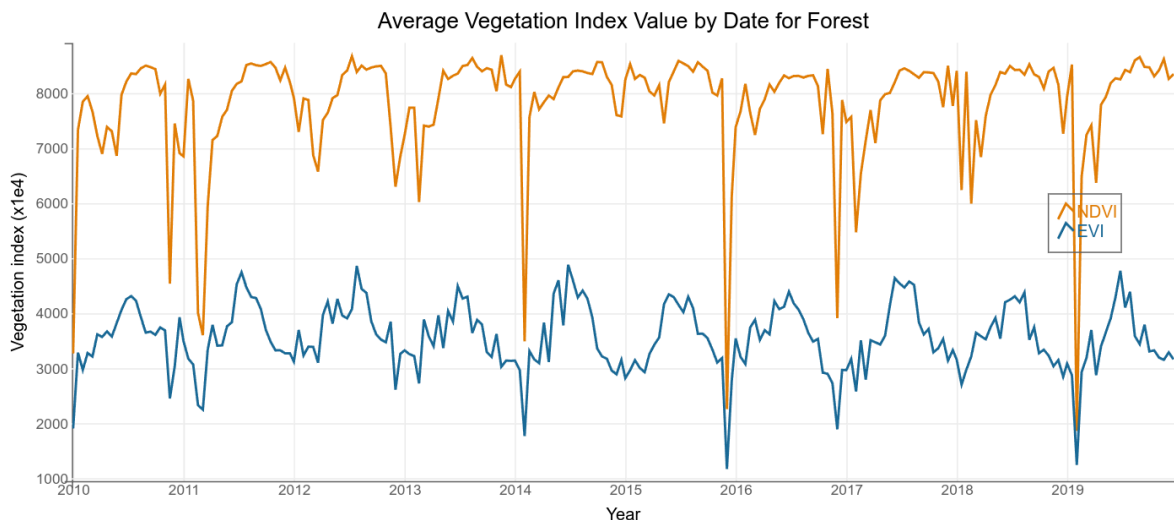


Figure 104: A time series chart showing the average vegetation index value by date for a forest region.

23.14.3.2. image_series_by_region

Shows NDVI time series by region, comparing desert, forest, and grassland areas. Each region has a unique series for easy comparison ([Figure 105](#)):

```
ecoregions = ee.FeatureCollection("projects/google/charts_feature_example")

veg_indices = (
    ee.ImageCollection("MODIS/061/MOD13A1")
    .filter(ee.Filter.date("2010-01-01", "2020-01-01"))
    .select(["NDVI"])
)

title = "Average NDVI Value by Date"
x_label = "Date"
y_label = "NDVI (x1e4)"
x_cols = "index"
y_cols = ["Desert", "Forest", "Grassland"]
colors = ["#f0af07", "#0f8755", "#76b349"]

fig = chart.image_series_by_region(
    veg_indices,
    regions=ecoregions,
    reducer=ee.Reducer.mean(),
    band="NDVI",
    scale=500,
    x_property="system:time_start",
    series_property="label",
    chart_type="LineChart",
    x_cols=x_cols,
    y_cols=y_cols,
    title=title,
    x_label=x_label,
    y_label=y_label,
    colors=colors,
    stroke_width=3,
    legend_location="bottom-left",
)
fig
```

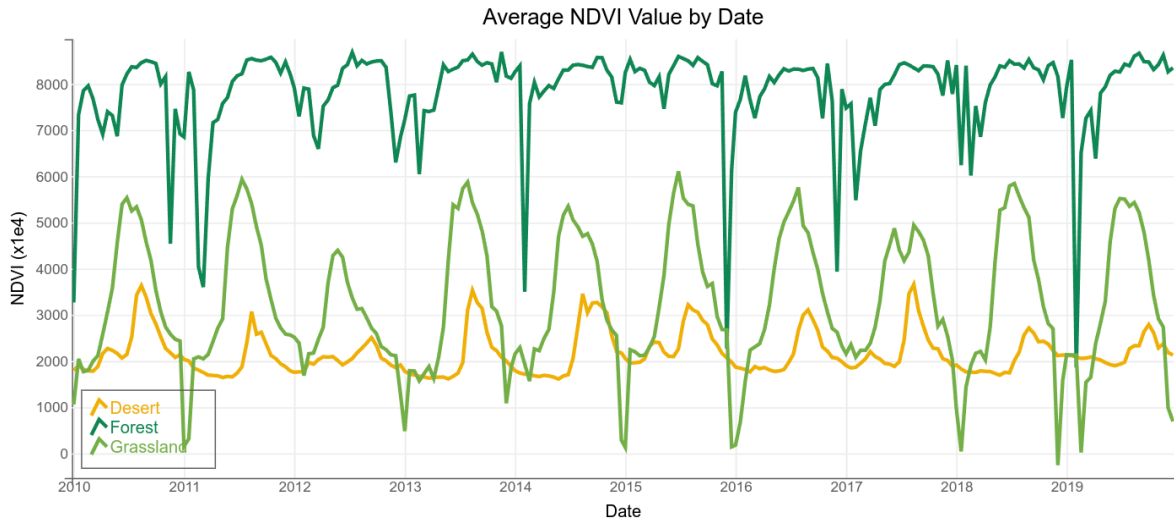


Figure 105: A time series chart showing the average NDVI value by date for different ecoregions.

23.14.3.3. image_doy_series

Plots average vegetation index by day of year for a specific region, showing seasonal vegetation patterns over a decade (Figure 106):

```
# Import the example feature collection and subset the grassland feature.
grassland = ee.FeatureCollection("projects/google/
charts_feature_example").filter(
    ee.Filter.eq("label", "Grassland")
)

# Load MODIS vegetation indices data and subset a decade of images.
veg_indices = (
    ee.ImageCollection("MODIS/061/MOD13A1")
    .filter(ee.Filter.date("2010-01-01", "2020-01-01"))
    .select(["NDVI", "EVI"])
)

title = "Average Vegetation Index Value by Day of Year for Grassland"
x_label = "Day of Year"
y_label = "Vegetation Index (x1e4)"
colors = ["#f0af07", "#0f8755"]

fig = chart.image_doy_series(
    image_collection=veg_indices,
    region=grassland,
    scale=500,
    chart_type="LineChart",
```

```

    title=title,
    x_label=x_label,
    y_label=y_label,
    colors=colors,
    stroke_width=5,
)
fig

```

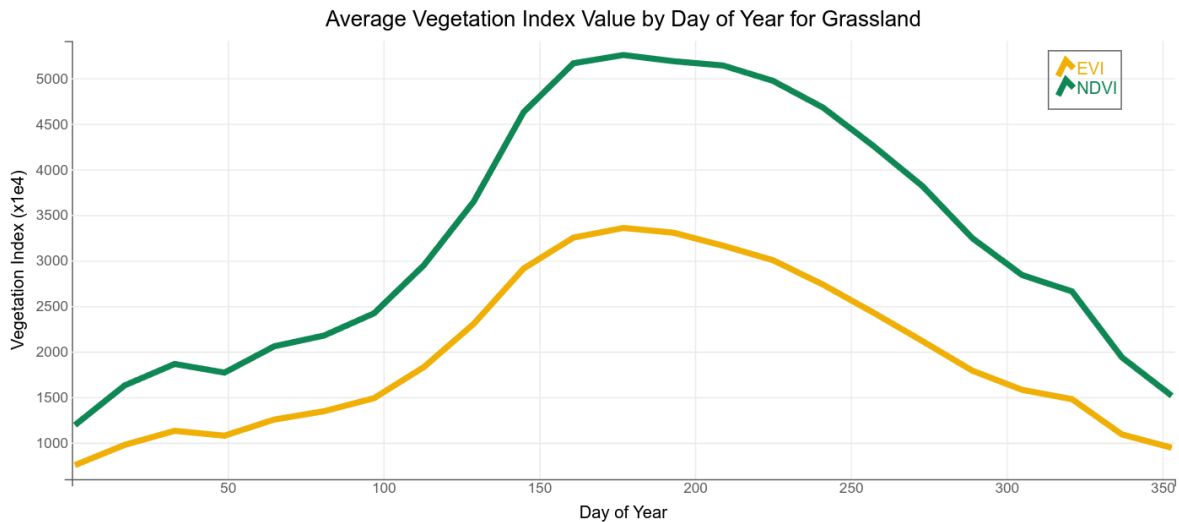


Figure 106: A time series chart showing the average NDVI value by day of year for a grassland region.

23.14.3.4. image_doy_series_by_year

Compares NDVI by day of year across two different years, showing how vegetation patterns vary between 2012 and 2019 (Figure 107):

```

# Import the example feature collection and subset the grassland feature.
grassland = ee.FeatureCollection("projects/google/
charts_feature_example").filter(
    ee.Filter.eq("label", "Grassland")
)

# Load MODIS vegetation indices data and subset years 2012 and 2019.
veg_indices = (
    ee.ImageCollection("MODIS/061/MOD13A1")
    .filter(
        ee.Filter.Or(
            ee.Filter.date("2012-01-01", "2013-01-01"),
            ee.Filter.date("2019-01-01", "2020-01-01"),
        )
    )
)

```

```

    )
    .select(["NDVI", "EVI"])
  )

  title = "Average Vegetation Index Value by Day of Year for Grassland"
  x_label = "Day of Year"
  y_label = "Vegetation Index (x1e4)"
  colors = ["#e37d05", "#1d6b99"]

  fig = chart.doy_series_by_year(
    veg_indices,
    band_name="NDVI",
    region=grassland,
    scale=500,
    chart_type="LineChart",
    colors=colors,
    title=title,
    x_label=x_label,
    y_label=y_label,
    stroke_width=5,
  )
  fig

```

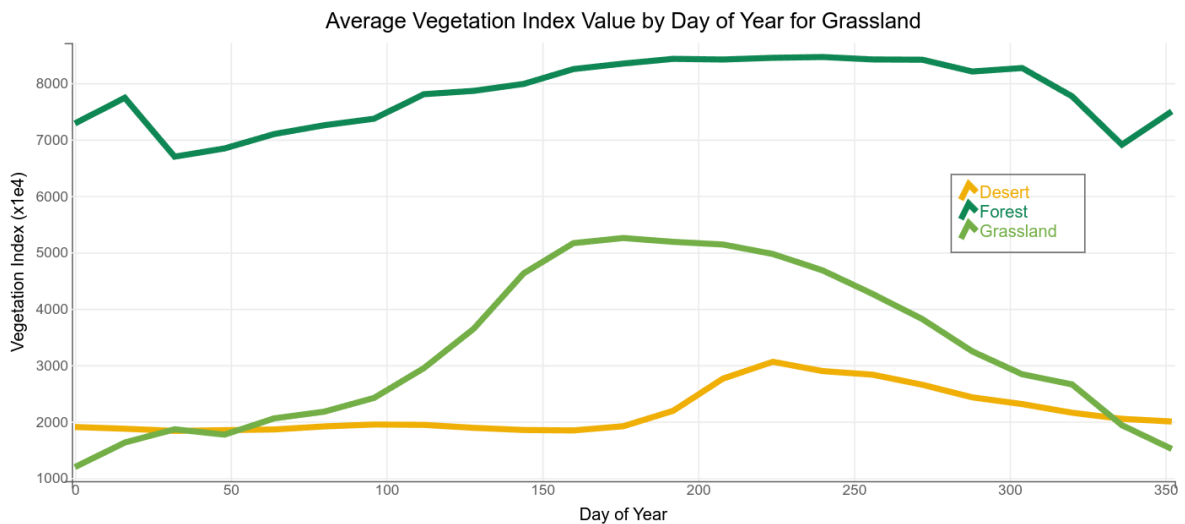


Figure 107: A time series chart showing the average NDVI value by day of year for a grassland region.

23.14.3.5. image_doy_series_by_region

Visualizes average NDVI by day of year across regions, showcasing seasonal changes for desert, forest, and grassland ecoregions (Figure 108):


```

# Import the example feature collection and subset the grassland feature.
ecoregions = ee.FeatureCollection("projects/google/charts_feature_example")

# Load MODIS vegetation indices data and subset a decade of images.
veg_indices = (
  ee.ImageCollection("MODIS/061/MOD13A1")
    .filter(ee.Filter.date("2010-01-01", "2020-01-01"))
    .select(["NDVI"])
)

title = "Average Vegetation Index Value by Day of Year for Grassland"
x_label = "Day of Year"
y_label = "Vegetation Index (x1e4)"
colors = ["#f0af07", "#0f8755", "#76b349"]

fig = chart.image_doy_series_by_region(
  veg_indices,
  "NDVI",
  ecoregions,
  region_reducer="mean",
  scale=500,
  year_reducer=ee.Reducer.mean(),
  start_day=1,
  end_day=366,
  series_property="label",
  stroke_width=5,
  chart_type="LineChart",
  title=title,
  x_label=x_label,
  y_label=y_label,
  colors=colors,
  legend_location="right",
)
fig

```

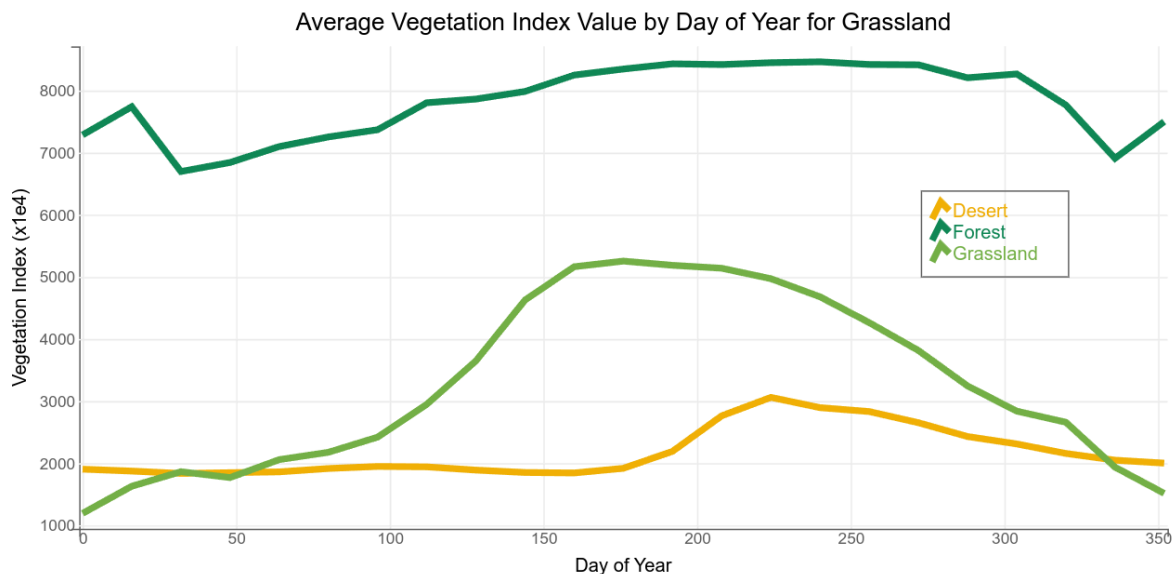


Figure 108: A time series chart showing the average NDVI value by day of year by ecoregion.

23.14.4. Charting Array and List

Array and list-based charts provide flexibility for visualizing computed data or exploring relationships between variables. These charts work with Earth Engine Arrays and Lists, which can contain extracted pixel values, statistical summaries, or computed results from Earth Engine operations. This approach is particularly useful for creating scatter plots, correlation analyses, and custom visualizations that don't fit standard chart templates.

23.14.4.1. Scatter Plot

Scatter plots are essential for exploring relationships between variables, identifying correlations, and understanding data structure. In remote sensing applications, scatter plots are commonly used to examine relationships between spectral bands, assess data quality, or identify distinct spectral signatures.

This example creates a scatter plot showing the relationships between different spectral bands in a forest area, helping to understand how different wavelengths of light are reflected by forest vegetation (Figure 109):

```
# Import the example feature collection and subset the forest feature.
forest = ee.FeatureCollection("projects/google/charts_feature_example").filter(
    ee.Filter.eq("label", "Forest")
)

# Define a MODIS surface reflectance composite.
modisSr = (
    ee.ImageCollection("MODIS/061/MOD09A1")
    .filter(ee.Filter.date("2018-06-01", "2018-09-01"))
)
```

```

        .select("sur_refl_b0[0-7]")
        .mean()
    )

    # Reduce MODIS reflectance bands by forest region; get a dictionary with
    # band names as keys, pixel values as lists.
    pixel_vals = modisSr.reduceRegion(
        **{"reducer": ee.Reducer.toList(), "geometry": forest.geometry(), "scale":
        2000}
    )

    # Convert NIR and SWIR value lists to an array to be plotted along the y-axis.
    y_values = pixel_vals.toArray(["sur_refl_b02", "sur_refl_b06"])

    # Get the red band value list; to be plotted along the x-axis.
    x_values = ee.List(pixel_vals.get("sur_refl_b01"))

    title = "Relationship Among Spectral Bands for Forest Pixels"
    colors = ["rgba(29,107,153,0.4)", "rgba(207,81,62,0.4)"]

    fig = chart.array_values(
        y_values,
        axis=1,
        x_labels=x_values,
        series_names=["NIR", "SWIR"],
        chart_type="ScatterChart",
        colors=colors,
        title=title,
        x_label="Red reflectance (x1e4)",
        y_label="NIR & SWIR reflectance (x1e4)",
        default_size=15,
        xlim=(0, 800),
    )
    fig

```

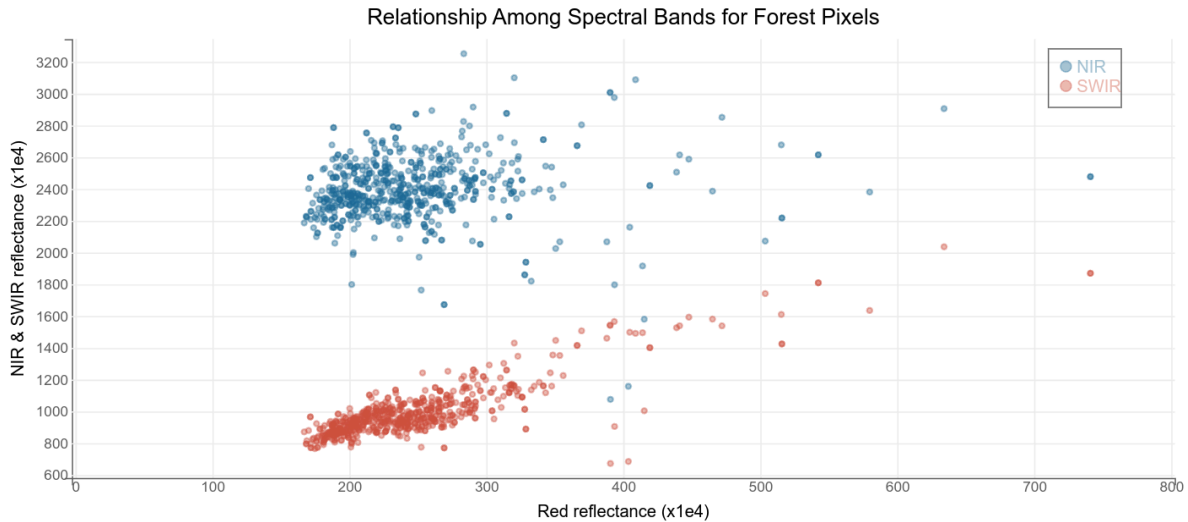


Figure 109: A scatter plot showing the relationship between different spectral bands in a forest area.

23.14.4.2. Transect Line Plot

Transect plots show how values change along a linear path, providing cross-sectional views of geographic phenomena. These plots are particularly valuable for examining topographic profiles, analyzing gradients across environmental boundaries, or investigating spatial patterns along specific routes or directions.

This example creates an elevation profile across the Olympic Peninsula, showing how terrain elevation varies from the Pacific coast to the interior mountains ([Figure 110](#)):

```
# Define a line across the Olympic Peninsula, USA.
transect = ee.Geometry.LineString([[-122.8, 47.8], [-124.5, 47.8]])

# Define a pixel coordinate image.
lat_lon_img = ee.Image.pixelLonLat()

# Import a digital surface model and add latitude and longitude bands.
elev_img = ee.Image("USGS/SRTMGL1_003").select("elevation").addBands(lat_lon_img)

# Reduce elevation and coordinate bands by transect line; get a dictionary with
# band names as keys, pixel values as lists.
elev_transect = elev_img.reduceRegion(
    reducer=ee.Reducer.toList(),
    geometry=transect,
    scale=1000,
)

# Get longitude and elevation value lists from the reduction dictionary.
lon = ee.List(elev_transect.get("longitude"))
elev = ee.List(elev_transect.get("elevation"))
```

```
# Sort the longitude and elevation values by ascending longitude.
lon_sort = lon.sort(lon)
elev_sort = elev.sort(lon)

fig = chart.array_values(
    elev_sort,
    x_labels=lon_sort,
    series_names=["Elevation"],
    chart_type="AreaChart",
    colors=["#1d6b99"],
    title="Elevation Profile Across Longitude",
    x_label="Longitude",
    y_label="Elevation (m)",
    stroke_width=5,
    fill="bottom",
    fill_opacities=[0.4],
    ylim=(0, 2500),
)
fig
```

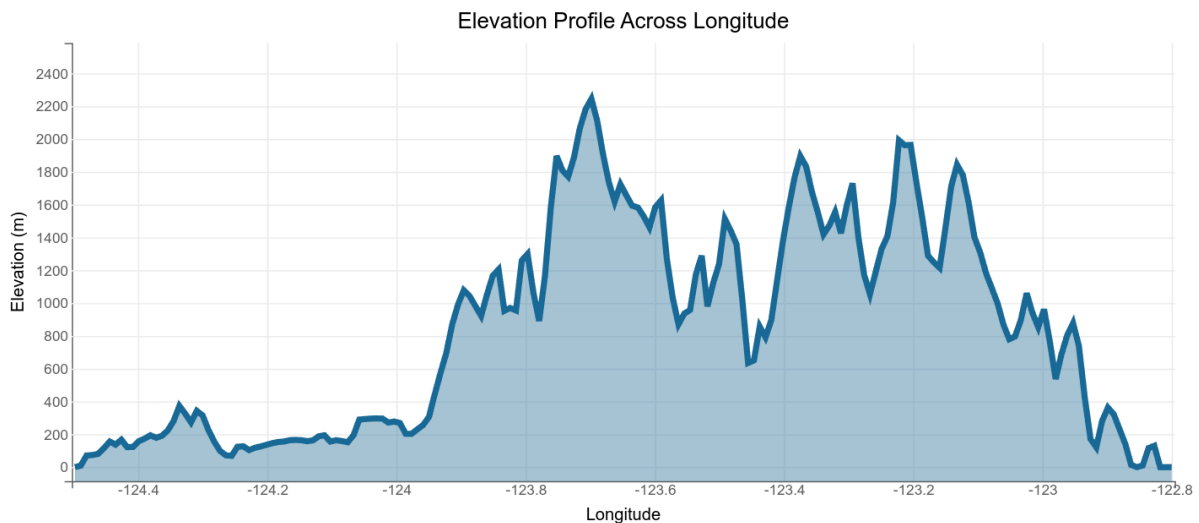


Figure 110: A chart showing the elevation profile across longitude.

23.14.4.3. Metadata Scatter Plot

Metadata analysis charts help assess data quality and understand dataset characteristics. By plotting different metadata attributes against each other, you can identify patterns, outliers, and relationships that inform data selection and quality control decisions.

This example creates a scatter plot of Landsat 8 image metadata, plotting cloud cover percentage against geometric accuracy measures to help identify high-quality images for analysis ([Figure 111](#)):

```

# Import a Landsat 8 collection and filter to a single path/row.
col = ee.ImageCollection("LANDSAT/LC08/C02/T1_L2").filter(
    ee.Filter.expression("WRS_PATH == 45 && WRS_ROW == 30")
)

# Reduce image properties to a series of lists; one for each selected property.
propVals = col.reduceColumns(
    reducer=ee.Reducer.toList().repeat(2),
    selectors=["CLOUD_COVER", "GEOMETRIC_RMSE_MODEL"],
).get("list")

# Get selected image property value lists; to be plotted along x and y axes.
x = ee.List(ee.List(propVals).get(0))
y = ee.List(ee.List(propVals).get(1))

colors = [geemap.hex_to_rgba("#96356f", 0.4)]
fig = chart.array_values(
    y,
    x_labels=x,
    series_names=["RMSE"],
    chart_type="ScatterChart",
    colors=colors,
    title="Landsat 8 Image Collection Metadata (045030)",
    x_label="Cloud cover (%)",
    y_label="Geometric RMSE (m)",
    default_size=15,
)
fig

```

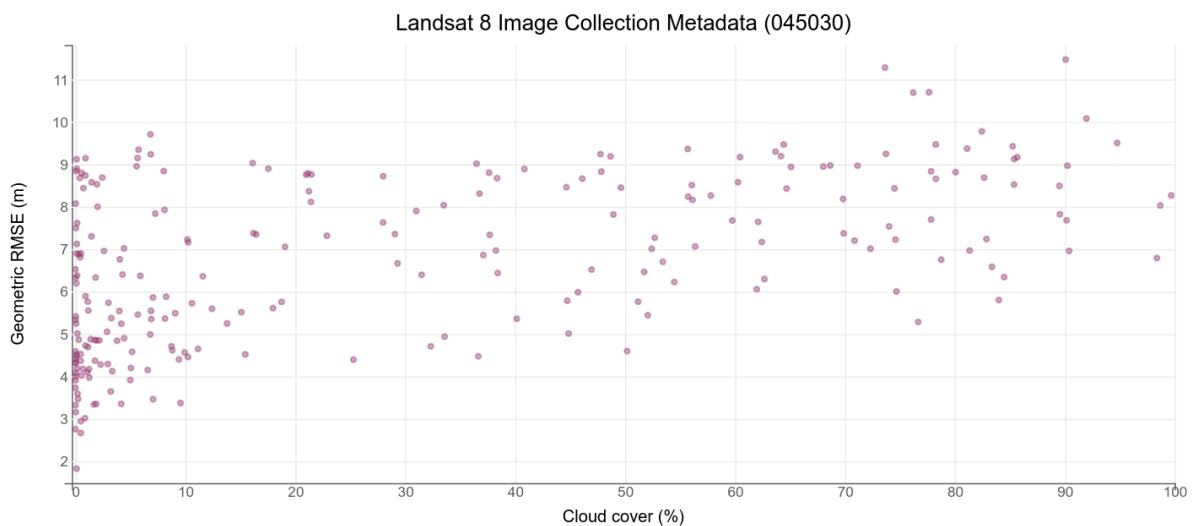


Figure 111: A scatter plot showing the relationship between cloud cover and geometric RMSE for Landsat 8 images.

23.14.4.4. Mapped Function Scatter & Line Plot

Plots a sine function as a line chart, mapping mathematical functions onto chart axes for visualization (Figure 112):

```
import math

start = -2 * math.pi
end = 2 * math.pi
points = ee.List.sequence(start, end, None, 50)

def sin_func(val):
    return ee.Number(val).sin()

values = points.map(sin_func)
fig = chart.array_values(
    values,
    points,
    chart_type="LineChart",
    colors=["#39a8a7"],
    title="Sine Function",
    x_label="radians",
    y_label="sin(x)",
    marker="circle",
)
fig
```

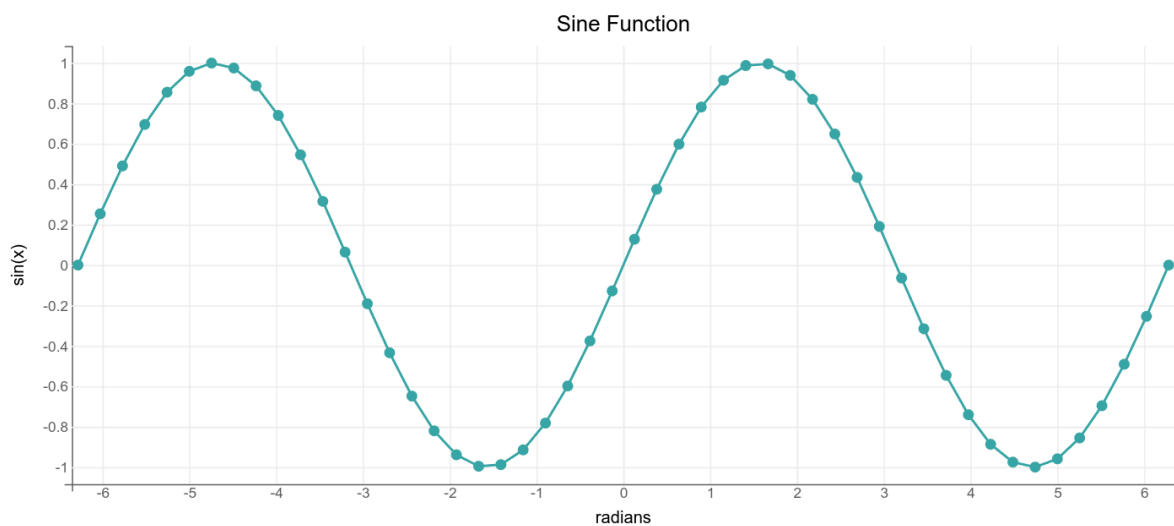


Figure 112: A line chart showing the sine function.

23.14.5. Charting Data Table

DataTable charts provide the most flexible approach to visualization by working with structured tabular data. This method is particularly useful when you need to combine Earth Engine data with external datasets, perform complex calculations, or create custom visualizations that don't fit standard templates. DataTables can be created manually or computed from Earth Engine operations.

23.14.5.1. Manual DataTable chart

Manual DataTable creation allows you to incorporate external data or create custom datasets for visualization. This approach is useful when combining Earth Engine analysis results with other data sources or when creating specific visualizations for reporting purposes.

This example demonstrates creating a simple chart from manually entered data about US state populations ([Figure 113](#)):

```
import pandas as pd

data = {
    "State": ["CA", "NY", "IL", "MI", "OR"],
    "Population": [37253956, 19378102, 12830632, 9883640, 3831074],
}

df = pd.DataFrame(data)
df
```

```
fig = chart.Chart(
    df,
    x_cols=["State"],
    y_cols=["Population"],
    chart_type="ColumnChart",
    colors=["#1d6b99"],
    title="State Population (US census, 2010)",
    x_label="State",
    y_label="Population",
)
fig
```

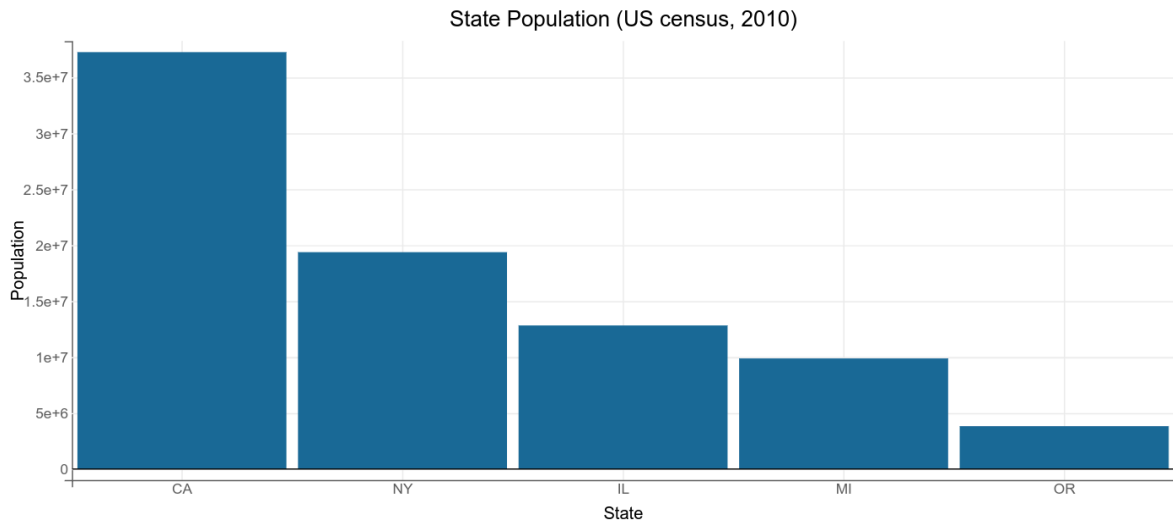



Figure 113: A chart showing the population of each state in the US.

23.14.5.2. Computed DataTable chart

Computed DataTables are created by processing Earth Engine data through reduction operations, allowing you to extract specific statistics and measurements for visualization. This approach provides complete control over data processing and enables complex analytical workflows that combine multiple datasets or calculation steps.

This example demonstrates computing a DataTable from MODIS vegetation indices data, extracting time series information for a forest region ([Figure 114](#)):

```
# Import the example feature collection and subset the forest feature.
forest = ee.FeatureCollection("projects/google/charts_feature_example").filter(
    ee.Filter.eq("label", "Forest")
)

# Load MODIS vegetation indices data and subset a decade of images.
veg_indices = (
    ee.ImageCollection("MODIS/061/MOD13A1")
    .filter(ee.Filter.date("2010-01-01", "2020-01-01"))
    .select(["NDVI", "EVI"])
)

# Build a feature collection where each feature has a property that represents
# a DataFrame row.

def aggregate(img):
    # Reduce the image to the mean of pixels intersecting the forest ecoregion.
    stat = img.reduceRegion(
```

```

    **{"reducer": ee.Reducer.mean(), "geometry": forest, "scale": 500}
)

# Extract the reduction results along with the image date.
date = geemap.image_date(img)
evi = stat.get("EVI")
ndvi = stat.get("NDVI")

# Make a list of observation attributes to define a row in the DataTable.
row = ee.List([date, evi, ndvi])

# Return the row as a property of an ee.Feature.
return ee.Feature(None, {"row": row})

reduction_table = veg_indices.map(aggregate)

# Aggregate the 'row' property from all features in the new feature collection
# to make a server-side 2-D list (DataTable).
data_table_server = reduction_table.aggregate_array("row")

# Define column names and properties for the DataTable. The order should
# correspond to the order in the construction of the 'row' property above.
column_header = ee.List(["Date", "EVI", "NDVI"])

# Concatenate the column header to the table.
data_table_server = column_header.cat(data_table_server)
data_table = chart.DataTable(data_table_server, date_column="Date")
data_table.head()

```

```

fig = chart.Chart(
    data_table,
    chart_type="LineChart",
    x_cols="Date",
    y_cols=["EVI", "NDVI"],
    colors=["#e37d05", "#1d6b99"],
    title="Average Vegetation Index Value by Date for Forest",
    x_label="Date",
    y_label="Vegetation index (x1e4)",
    stroke_width=3,
    legend_location="right",
)
fig

```

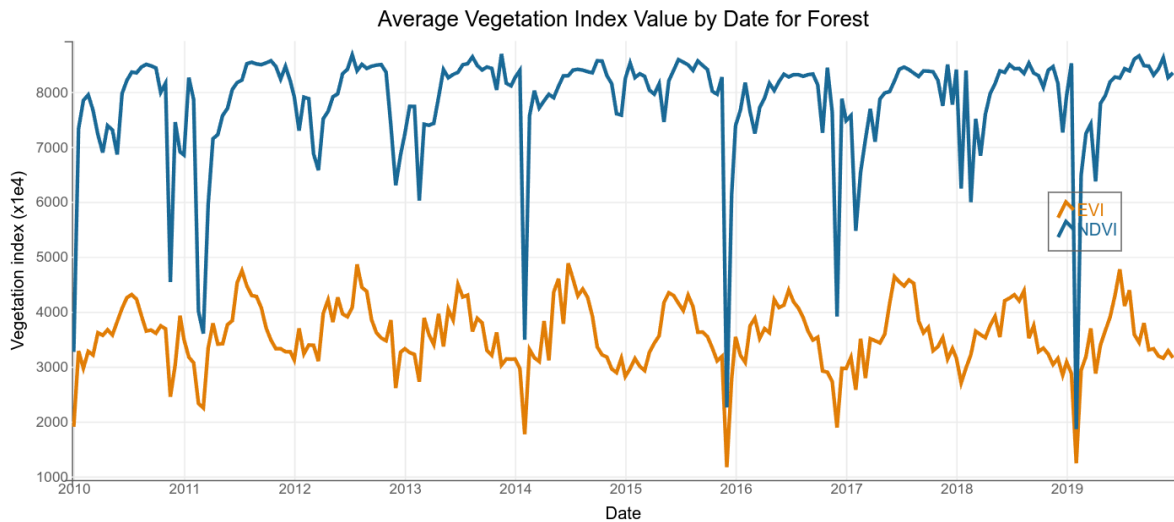


Figure 114: A chart showing the average NDVI value by date for a forest region.

23.14.5.3. Interval chart

Interval charts are specialized visualizations that display statistical ranges and uncertainty in data. These charts show not just central tendencies (like means or medians) but also the variability around these values, providing a more complete picture of data distribution. Interval charts are particularly valuable for climate analysis, where understanding both average conditions and variability is crucial.

This example creates an interval chart showing annual NDVI patterns with inter-annual variance, revealing both typical seasonal cycles and the range of variation across multiple years ([Figure 115](#)):

```
# Define a point to extract an NDVI time series for.
geometry = ee.Geometry.Point([-121.679, 36.479])

# Define a band of interest (NDVI), import the MODIS vegetation index dataset,
# and select the band.
band = "NDVI"
ndvi_col = ee.ImageCollection("MODIS/061/MOD13Q1").select(band)

# Map over the collection to add a day of year (doy) property to each image.

def set_doy(img):
    doy = ee.Date(img.get("system:time_start")).getRelative("day", "year")
    # Add 8 to day of year number so that the doy label represents the middle of
    # the 16-day MODIS NDVI composite.
    return img.set("doy", ee.Number(doy).add(8))

ndvi_col = ndvi_col.map(set_doy)
```

```
# Join all coincident day of year observations into a set of image collections.
distinct_doy = ndvi_col.filterDate("2013-01-01", "2014-01-01")
filter = ee.Filter.equals(**{"leftField": "doy", "rightField": "doy"})
join = ee.Join.saveAll("doy_matches")
join_col = ee.ImageCollection(join.apply(distinct_doy, ndvi_col, filter))
```

```
# Calculate the absolute range, interquartile range, and median for the set
# of images composing each coincident day observation group. The result is
# an image collection with an image representative per unique day observation
# with bands that describe the 0, 25, 50, 75, 100 percentiles for the set of
# coincident day images.
```

```
def cal_percentiles(img):
    doyCol = ee.ImageCollection.fromImages(img.get("doy_matches"))

    return doyCol.reduce(
        ee.Reducer.percentile([0, 25, 50, 75, 100], ["p0", "p25", "p50", "p75",
"p100"]))
    ).set({"doy": img.get("doy")})
```

```
comp = ee.ImageCollection(join_col.map(cal_percentiles))
```

```
# Extract the inter-annual NDVI doy percentile statistics for the
# point of interest per unique doy representative. The result is
# is a feature collection where each feature is a doy representative that
# contains a property (row) describing the respective inter-annual NDVI
# variance, formatted as a list of values.
```

```
def order_percentiles(img):
    stats = ee.Dictionary(
        img.reduceRegion(
            **{"reducer": ee.Reducer.first(), "geometry": geometry, "scale": 250}
        )
    )
```

```
# Order the percentile reduction elements according to how you want columns
# in the DataTable arranged (x-axis values need to be first).
```

```
row = ee.List(
    [
        img.get("doy"),
        stats.get(band + "_p50"),
        stats.get(band + "_p0"),
        stats.get(band + "_p25"),
        stats.get(band + "_p75"),
```

```

        stats.get(band + "_p100"),
    ]
)

# Return the row as a property of an ee.Feature.
return ee.Feature(None, {"row": row})

reduction_table = comp.map(order_percentiles)

# Aggregate the 'row' properties to make a server-side 2-D array (DataTable).
data_table_server = reduction_table.aggregate_array("row")

# Define column names and properties for the DataTable. The order should
# correspond to the order in the construction of the 'row' property above.
column_header = ee.List(["DOY", "median", "p0", "p25", "p75", "p100"])

# Concatenate the column header to the table.
data_table_server = column_header.cat(data_table_server)
df = chart.DataTable(data_table_server)
df.head()

```

```

fig = chart.Chart(
    df,
    chart_type="IntervalChart",
    x_cols="DOY",
    y_cols=["p0", "p25", "median", "p75", "p100"],
    title="Annual NDVI Time Series with Inter-Annual Variance",
    x_label="Day of Year",
    y_label="Vegetation index (x1e4)",
    stroke_width=1,
    fill="between",
    fill_colors=["#b6d1c6", "#83b191", "#83b191", "#b6d1c6"],
    fill_opacities=[0.6] * 4,
    labels=["p0", "p25", "median", "p75", "p100"],
    display_legend=True,
    legend_location="top-right",
    ylim=(0, 10000),
)
fig

```

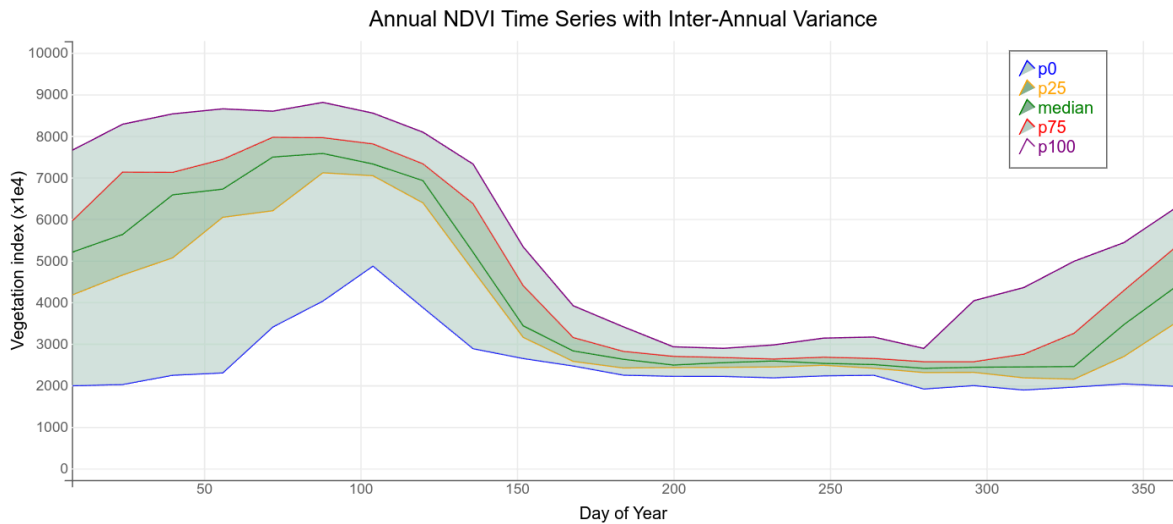


Figure 115: A chart showing the annual NDVI time series with inter-annual variance.

23.15. Key Takeaways

This chapter introduced cloud-based geospatial analysis using **Google Earth Engine** and **geemap**, representing a fundamental shift from traditional desktop GIS to cloud computing. The key concepts and skills covered include:

Platform Fundamentals: Earth Engine provides access to over 1,000 geospatial datasets and 100+ petabytes of analysis-ready data through cloud computing, eliminating the need for local data storage and high-end hardware. The geemap package bridges Earth Engine’s JavaScript API with Python’s ecosystem, enabling familiar Jupyter notebook workflows.

Data Types and Processing: Earth Engine organizes geospatial data into fundamental types—Images for raster data, ImageCollections for time series, and Features/FeatureCollections for vector data. Understanding these server-side objects is crucial for effective cloud-based analysis, where data remains on Google’s servers while computation instructions are sent remotely.

Interactive Visualization: Geemap provides powerful interactive mapping capabilities with customizable basemaps, layer management, and real-time parameter adjustment. Tools like the inspector, layer editor, and drawing controls transform static maps into dynamic exploration environments for rapid data discovery and analysis.

Temporal Analysis: The chapter demonstrated creating compelling timelapse animations from multiple satellite platforms (Landsat, Sentinel-2, MODIS, GOES), each optimized for different temporal scales and phenomena. These visualizations compress years of observations into seconds, revealing patterns of change invisible in static imagery.

Statistical Visualization: Earth Engine data can be transformed into various chart types—time series, histograms, scatter plots, and statistical summaries. These visualizations complement spatial maps by revealing temporal trends, data distributions, and quantitative relationships between variables.

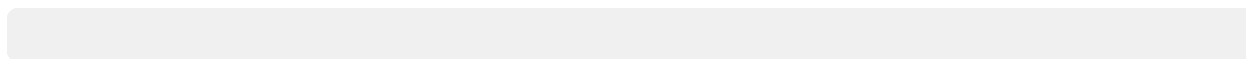
Analytical Workflows: The chapter covered essential operations including data filtering, zonal statistics, map algebra, and data export. These foundational skills enable complex analytical workflows that leverage Earth Engine’s computational power for scientific research and operational applications.

The paradigm shift to cloud-based geospatial analysis opens new possibilities for research and analysis at scales previously accessible only to major institutions, democratizing access to planetary-scale environmental monitoring and analysis capabilities.

23.16. Exercises

23.16.1. Exercise 1: Visualizing DEM Data

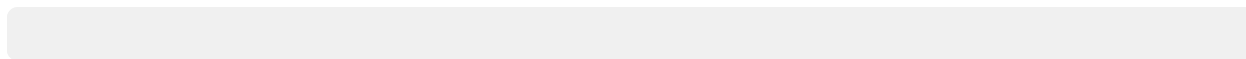
Find a DEM dataset in the [Earth Engine Data Catalog](#) and clip it to a specific area (e.g., your country, state, or city). Display it with an appropriate color palette. For example, the sample map below shows the DEM of the state of Colorado.



23.16.2. Exercise 2: Cloud-Free Composite with Sentinel-2 or Landsat

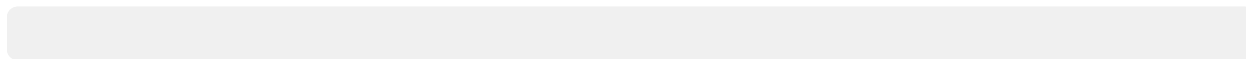
Use Sentinel-2 or Landsat-9 data to create a cloud-free composite for a specific year in a region of your choice.

Use [Sentinel-2](#) or [Landsat-9 data](#) to create a cloud-free composite for a specific year in a region of your choice. Display the imagery on the map with a proper band combination. For example, the sample map below shows a cloud-free false-color composite of Sentinel-2 imagery of the year 2021 for the state of Colorado.



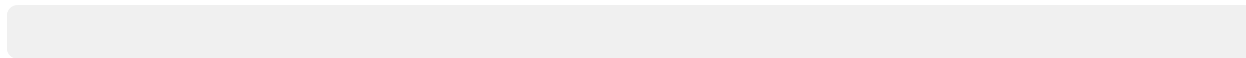
23.16.3. Exercise 3: Visualizing NAIP Imagery

Use [NAIP](#) imagery to create a cloud-free imagery for a U.S. county of your choice. For example, the sample map below shows a cloud-free true-color composite of NAIP imagery for Knox County, Tennessee. Keep in mind that there might be some counties with the same name in different states, so make sure to select the correct county for the selected state.



23.16.4. Exercise 4: Visualizing Watershed Boundaries

Visualize the [USGS Watershed Boundary Dataset](#) with outline color only, no fill color.



23.16.5. Exercise 5: Visualizing Land Cover Change

Use the [USGS National Land Cover Database](#) and [US Census States](#) to create a split-panel map for visualizing land cover change (2001-2019) for a US state of your choice. Make sure you add the NLCD legend to the map.

23.16.6. Exercise 6: Creating a Landsat Timelapse Animation

Generate a timelapse animation using Landsat data to show changes over time for a selected region.

Chapter 24. Hyperspectral Data Visualization with HyperCoast

24.1. Introduction

Hyperspectral remote sensing captures detailed spectral information across hundreds of narrow wavelength bands, representing one of the most advanced forms of Earth observation technology. Unlike traditional multispectral imagery that typically uses 3-10 broad bands, hyperspectral sensors record reflectance values across 200-300 or more contiguous spectral bands, creating a continuous spectrum for each pixel.

This rich spectral information enables scientists to identify and analyze materials, vegetation types, water quality parameters, and geological features with unprecedented precision. Each pixel contains a complete spectral signature that can distinguish between materials that appear identical in traditional RGB or multispectral imagery.

NASA's Earth Surface Mineral Dust Source Investigation (EMIT)¹¹⁷ instrument on the International Space Station provides leading hyperspectral data. While designed to map mineral composition of dust source regions, EMIT's applications extend to vegetation monitoring, water quality assessment, and environmental change detection.

The [HyperCoast](https://hypercoast.org)¹¹⁸ Python package supports reading and visualizing hyperspectral data from various missions, including [AVIRIS](https://aviris.jpl.nasa.gov)¹¹⁹, [NEON](https://data.neonscience.org/data-products/DP3.30006.001)¹²⁰, [PACE](https://pace.gsfc.nasa.gov)¹²¹, [EMIT](https://tinyurl.com/nasa-desis), and [DESIS](https://ecostress.jpl.nasa.gov)¹²², along with datasets like [ECOSTRESS](https://ecostress.jpl.nasa.gov)¹²³. Users can interactively explore hyperspectral data, extract spectral signatures, modify band combinations and colormaps, visualize data in 3D, and perform interactive slicing and thresholding operations. By leveraging the [earthaccess](https://github.com/nsidc/earthaccess)¹²⁴ package, HyperCoast provides tools for searching NASA's hyperspectral data archives. This makes HyperCoast a versatile tool for working with hyperspectral data globally, with particular strength in coastal regions.

24.2. Learning Objectives

By the end of this chapter, you will be able to:

- Understand the fundamental principles of hyperspectral remote sensing and its advantages over multispectral imaging
- Set up the HyperCoast environment and authenticate with NASA Earthdata services
- Search for and identify relevant EMIT hyperspectral datasets using both programmatic and interactive methods
- Download and read hyperspectral data files using HyperCoast functions
- Visualize hyperspectral data on interactive maps with custom band combinations
- Extract and analyze spectral profiles from hyperspectral imagery
- Create three-dimensional visualizations of hyperspectral data cubes
- Use interactive widgets for data exploration through slicing and thresholding techniques

¹¹⁷<https://earth.jpl.nasa.gov/emit>

¹¹⁸<https://hypercoast.org>

¹¹⁹<https://aviris.jpl.nasa.gov>

¹²⁰<https://data.neonscience.org/data-products/DP3.30006.001>

¹²¹<https://pace.gsfc.nasa.gov>

¹²²<https://tinyurl.com/nasa-desis>

¹²³<https://ecostress.jpl.nasa.gov>

¹²⁴<https://github.com/nsidc/earthaccess>

24.3. Environment Setup

Before working with hyperspectral data, we need to install the HyperCoast package along with its dependencies. Uncomment and run the following cell to install the required packages.

```
# %pip install hypercoast pygis
```

Once installed, we can import the HyperCoast library, which provides all the functions we'll need for this chapter.

```
import hypercoast
```

Accessing NASA's hyperspectral data requires authentication through the Earthdata Login system. This system ensures proper access credentials and helps NASA track data usage for scientific purposes. Registration is free and provides access to vast archives of Earth observation data.

To download and access data, create an Earthdata login at urs.earthdata.nasa.gov¹²⁵. Once registered, run the following cell and enter your NASA Earthdata credentials.

```
hypercoast.nasa_earth_login()
```

24.4. Finding Hyperspectral Data

The first step in hyperspectral analysis is identifying and locating relevant datasets. HyperCoast provides both programmatic and interactive methods for searching EMIT data archives, allowing you to find datasets matching your spatial and temporal requirements.

24.4.1. Programmatic Search

The programmatic approach is ideal when you know the specific geographic area and time period of interest. This method returns structured results that can be easily processed and filtered. Specify the bounding box and time range, setting `count=-1` to return all results or `count=10` for the first 10 results.

```
results, gdf = hypercoast.search_emit(  
    bbox=(-83, 25, -81, 28),  
    temporal=("2024-04-01", "2024-05-16"),  
    count=10, # use -1 to return all datasets  
    return_gdf=True,  
)
```

The search function returns two objects: a list of dataset results and a GeoDataFrame containing spatial footprints. Bounding box coordinates use longitude-latitude format (west, south, east, north), while temporal ranges use ISO date format strings.

Let's plot the footprints of the returned datasets on a map (see [Figure 116](#)):

¹²⁵<https://urs.earthdata.nasa.gov>

```
gdf.explore()
```

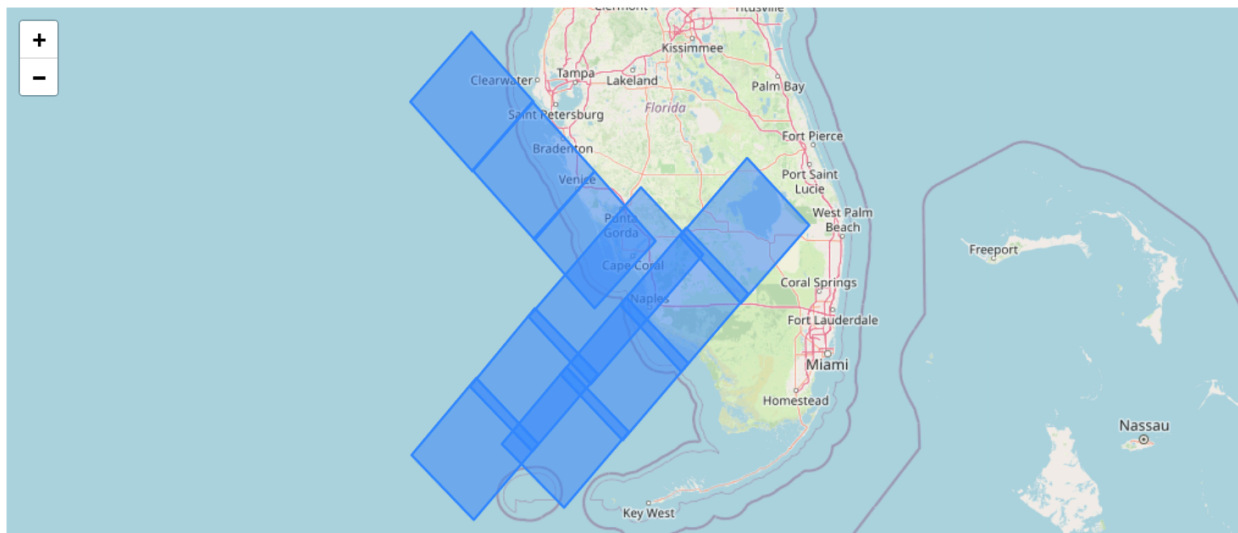


Figure 116: The EMIT image footprints covering the Florida coast.

This interactive map shows the spatial coverage of each EMIT dataset matching your search criteria. The footprints help you understand data availability and identify datasets that best cover your area of interest.

Run the following cell to download the first dataset. Note that downloads may take time as hyperspectral data files are large (typically 1-2 GB). Data is stored in the `data` directory.

```
hypercoast.download_emit(results[:1], out_dir="data")
```

24.4.2. Interactive Search

For exploratory analysis or when you're uncertain about specific coordinates, the interactive search method provides an intuitive way to discover hyperspectral data. You can pan and zoom to your area of interest, set temporal constraints through the search dialog, and immediately see results overlaid on the map (see [Figure 117](#)). This visual approach is particularly useful for understanding data availability and coverage patterns.

```
m = hypercoast.Map(center=[30.0262, -90.1345], zoom=8)
m.search_emit()
m
```

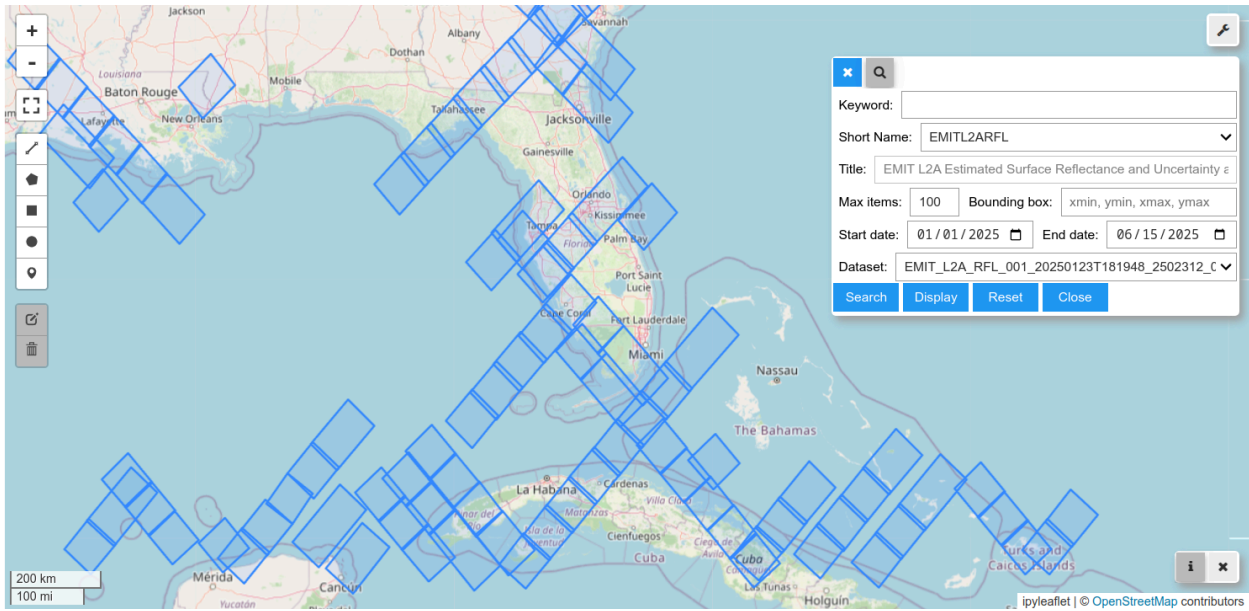


Figure 117: The interactive search GUI for the EMIT dataset.

Search results are automatically stored in the map object as a `GeoDataFrame`, allowing you to examine metadata and attributes of each dataset programmatically. Run the following cell to see the first few rows.

```
if hasattr(m, "_NASA_DATA_GDF"):
    display(m._NASA_DATA_GDF.head())
```

You can download the first dataset from search results by running the following cell. Note that downloads may take time as the data files are large and stored in the `data` directory.

```
hypercoast.download_emit(results[:1], out_dir="data")
```

24.5. Downloading Hyperspectral Data

For this demonstration, we'll work with a sample EMIT dataset representing typical hyperspectral data characteristics. This dataset covers a coastal area with diverse land cover types, making it ideal for exploring hyperspectral analysis capabilities.

Let's download a sample EMIT dataset from GitHub:

```
url = "https://github.com/opengeos/datasets/releases/download/hypercoast/EMIT_L2A_RFL_001_20240404T161230_2409511_009.nc"
filepath = "data/EMIT_L2A_RFL_001_20240404T161230_2409511_009.nc"
hypercoast.download_file(url, filepath, quiet=True)
```

The dataset filename follows NASA's naming convention, encoding important metadata including processing level (L2A), product type (RFL for reflectance), acquisition date, and orbit information. Understanding these conventions helps organize and manage hyperspectral data collections.

24.6. Reading Hyperspectral Data

Once downloaded, hyperspectral data must be read and processed into a format suitable for analysis. HyperCoast handles the complexities of EMIT data format and structure, providing a clean interface to the underlying spectral information.

Read the downloaded EMIT data and process it as an `xarray.Dataset` with 285 bands:

```
dataset = hypercoast.read_emit(filepath)
dataset
```

xarray.Dataset

► Dimensions:	(latitude: 1894, longitude: 2227, wavelength: 285)		
▼ Coordinates:			
wavelength	(wavelength)	float32	381.0 388.4 ... 2.486e+03 2.493e...
fwhm	(wavelength)	float32	...
good_wavelength	(wavelength)	float32	...
latitude	(latitude)	float64	30.37 30.37 30.36 ... 29.34 29.34
longitude	(longitude)	float64	-90.54 -90.54 ... -89.33 -89.33
elev	(latitude, longitude)	float32	nan nan nan nan ... nan nan na...
spatial_ref	()	int64	0
▼ Data variables:			
reflectance	(latitude, longitude, wavelength)	float32	nan nan nan nan ... nan nan na...
► Indexes:	(3)		
► Attributes:	(40)		

The resulting xarray Dataset provides structured representation of hyperspectral data, including coordinates, data variables, and attributes. The 285 spectral bands span visible through shortwave infrared portions of the electromagnetic spectrum, typically covering wavelengths from approximately 380 to 2500 nanometers. Each band represents a narrow spectral slice, enabling detailed analysis of surface materials.

24.7. Visualizing Hyperspectral Data

Effective visualization is crucial for understanding and interpreting hyperspectral data. While the human eye can only perceive three color channels (red, green, blue), hyperspectral data contains hundreds of bands. Therefore, we must carefully select which bands to display and how to combine them for meaningful visualization.

The most common approach creates false-color composites by assigning different spectral bands to red, green, and blue display channels. By selecting bands that highlight different material properties, we can create visualizations revealing features invisible to the naked eye.

Run the following cell to create a false-color composite of the EMIT dataset and display it on a map. The `Map.add_emit` method automatically creates a composite based on selected bands (Figure 118), and `Map.add("spectral")` adds an interactive GUI for changing band combinations and extracting spectral signatures.

```

m = hypercoast.Map()
m.add_basemap("SATELLITE")
m.add_emit(dataset, wavelengths=[1000, 600, 500], vmin=0, vmax=0.3,
layer_name="EMIT")
m.add("spectral")
m

```

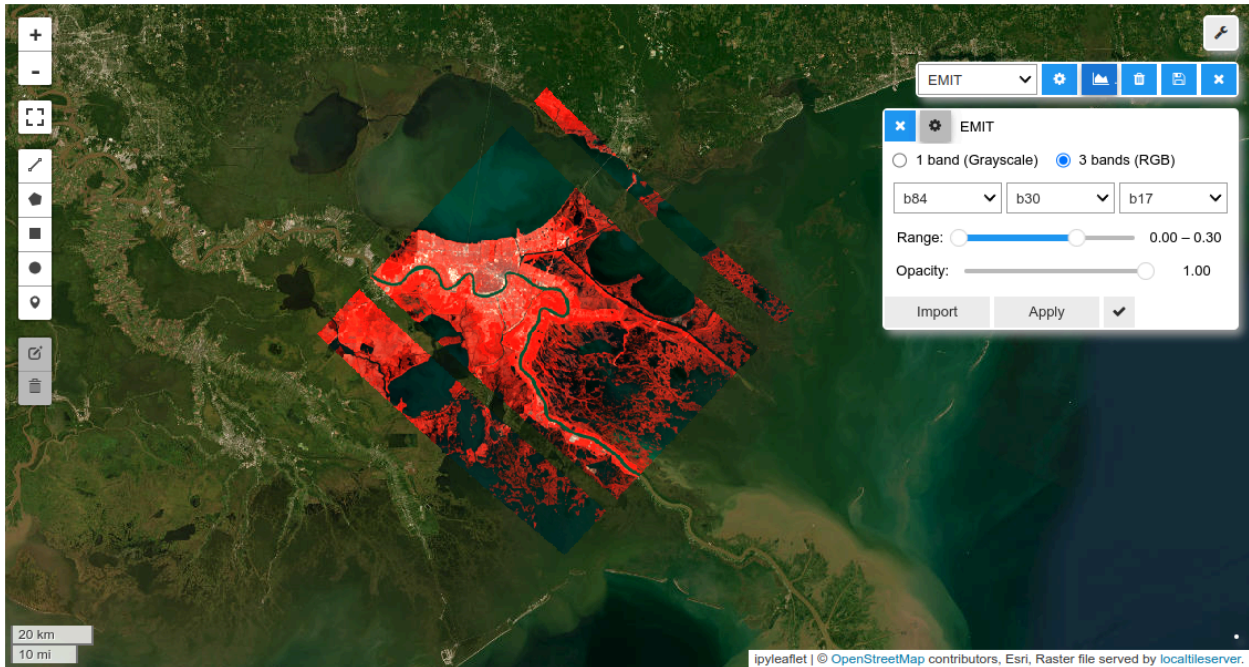


Figure 118: The interactive GUI for changing the band combination for the EMIT dataset.

This interactive visualization demonstrates several key capabilities. The wavelength selection (1000, 600, 500 nm) creates a false-color composite enhancing vegetation and water features. The spectral profiling tool allows you to click any pixel to extract its complete spectral signature, revealing detailed reflectance characteristics across all wavelengths (see [Figure 119](#)). This capability is fundamental to hyperspectral analysis, enabling material identification and quantitative analysis based on spectral properties.

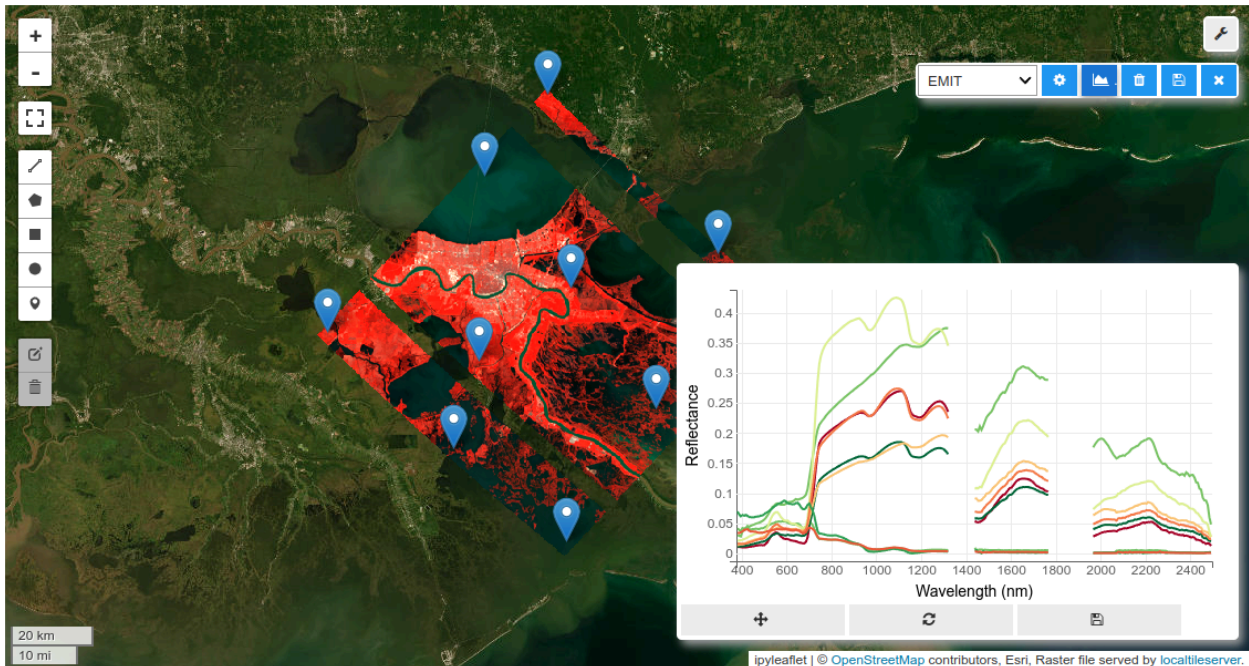


Figure 119: The interactive GUI for extracting spectral signatures from the EMIT dataset.

The extracted spectral signature can be saved to a CSV file for further analysis. Click the “Save” button on the toolbar to save the signature.

24.8. Creating Image Cubes

Hyperspectral data is inherently three-dimensional, with two spatial dimensions (x, y) and one spectral dimension (wavelength). Visualizing this data as a 3D cube helps understand the relationship between spatial and spectral information, providing insights difficult to obtain from traditional 2D visualizations.

First, select a subset of the data to avoid nodata areas.

```
ds = dataset.sel(longitude=slice(-90.1482, -89.7321), latitude=slice(30.0225,
29.7451))
```

Subsetting the data serves multiple purposes: reducing computational requirements, focusing analysis on areas with valid data, and eliminating edge effects that can occur in satellite imagery. The selected area should contain diverse land cover types to demonstrate hyperspectral data’s spectral discrimination capabilities.

Let’s visualize the EMIT data in 3D with an RGB image overlaid on top of the 3D plot (see [Figure 120](#)).

```
p = hypercoast.image_cube(
    ds,
    variable="reflectance",
    cmap="jet",
    clim=(0, 0.4),
```

```

rgb_wavelengths=[1000, 700, 500],
rgb_gamma=2,
title="EMIT Reflectance",
)
p.show()

```

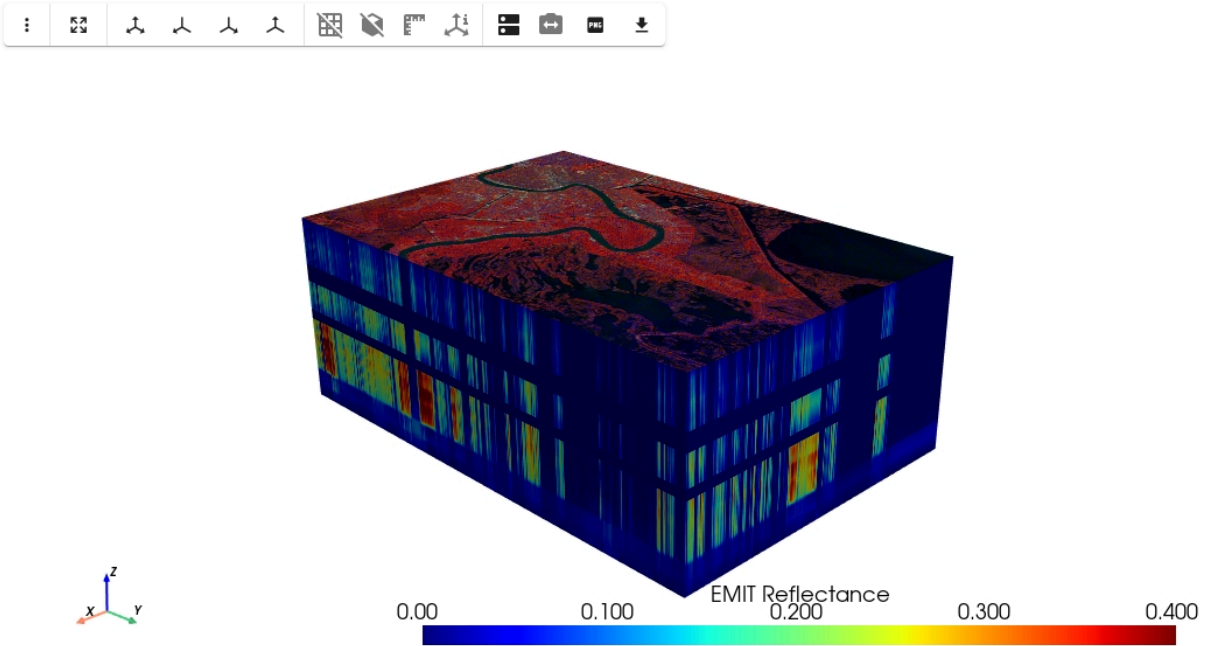


Figure 120: The 3D image cube of the EMIT dataset.

The 3D visualization combines spatial and spectral information in an intuitive format. The RGB overlay provides spatial context, while the underlying cube structure represents the spectral dimension. Color mapping and intensity scaling highlight variations in reflectance values across different wavelengths and spatial locations. Drag your mouse and scroll wheel over the plot to explore the data.

Important note: The 3D visualization is not supported in the Google Colab environment.

24.9. Interactive Slicing

Interactive slicing allows you to explore the spectral dimension of hyperspectral data by cutting through the data cube at different wavelength positions. This technique helps identify which wavelengths are most sensitive to particular materials or surface features.

First, select a subset of the data for demonstration purposes.

```

ds = dataset.sel(longitude=slice(-90.05, -89.99), latitude=slice(30.00, 29.93))

```

A smaller spatial subset ensures responsive interaction while maintaining sufficient detail for meaningful analysis. The selected area should ideally contain contrasting materials or land cover types to demonstrate spectral discrimination. Run the following cell to create a smaller subset and visualize it (Figure 121).


```

p = hypercoast.image_cube(
    ds,
    variable="reflectance",
    cmap="jet",
    clim=(0, 0.5),
    rgb_wavelengths=[1000, 700, 500],
    rgb_gamma=2,
    title="EMIT Reflectance",
    widget="plane",
)
p.add_text("Band slicing", position="upper_right", font_size=14)
p.show()

```

Band slicing

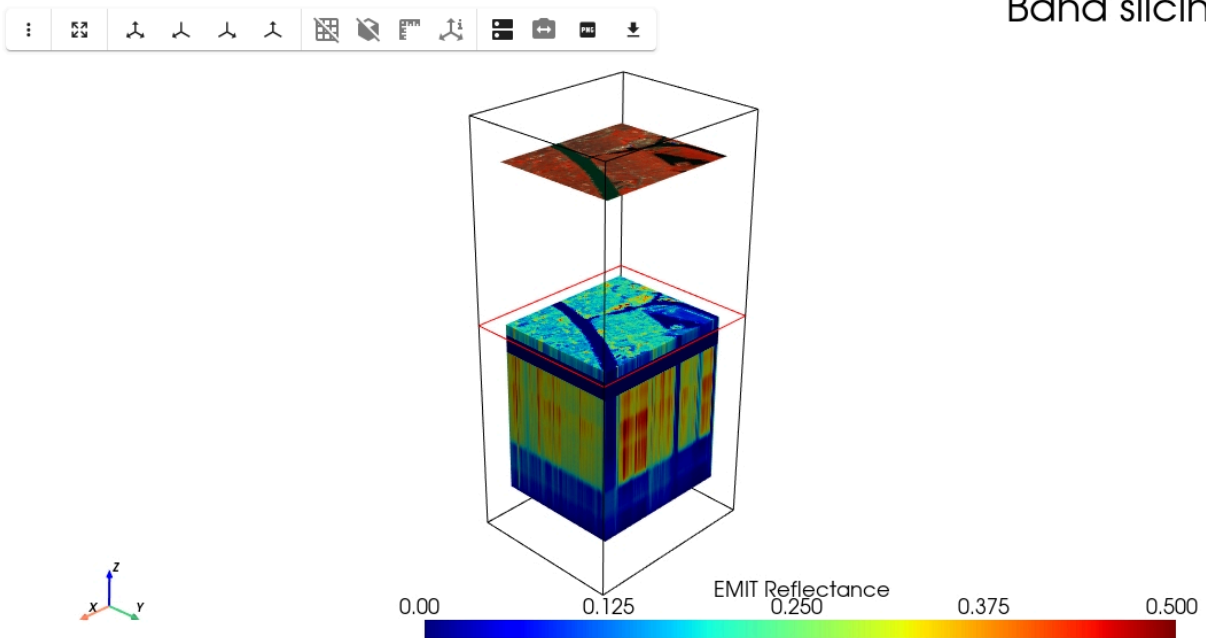


Figure 121: The interactive slicing widget for the EMIT dataset.

The slicing widget provides an intuitive way to navigate through the spectral dimension. As you move the slicing plane up and down, you're viewing different wavelength "slices" of the hyperspectral cube. This interaction reveals how different materials respond to various wavelengths, helping identify optimal bands for specific applications. Drag the plane up and down to slice the data in 3D.

The interactive slicing capability demonstrates how spectral information varies across wavelengths. Some features may be prominent in certain spectral ranges while invisible in others, highlighting the importance of multi-band analysis in hyperspectral remote sensing.

24.10. Interactive Thresholding

Thresholding is a fundamental technique for isolating pixels with specific reflectance characteristics. In hyperspectral data, thresholding can help identify materials with known spectral properties or highlight areas with unusual spectral signatures warranting further investigation.

Run the following cell to display the interactive thresholding widget (see [Figure 122](#)). Note that this function does not work in the Google Colab environment.

```
p = hypercoast.image_cube(  
    ds,  
    variable="reflectance",  
    cmap="jet",  
    clim=(0, 0.5),  
    rgb_wavelengths=[1000, 700, 500],  
    rgb_gamma=2,  
    title="EMIT Reflectance",  
    widget="threshold",  
)  
p.add_text("Thresholding", position="upper_right", font_size=14)  
p.show()
```

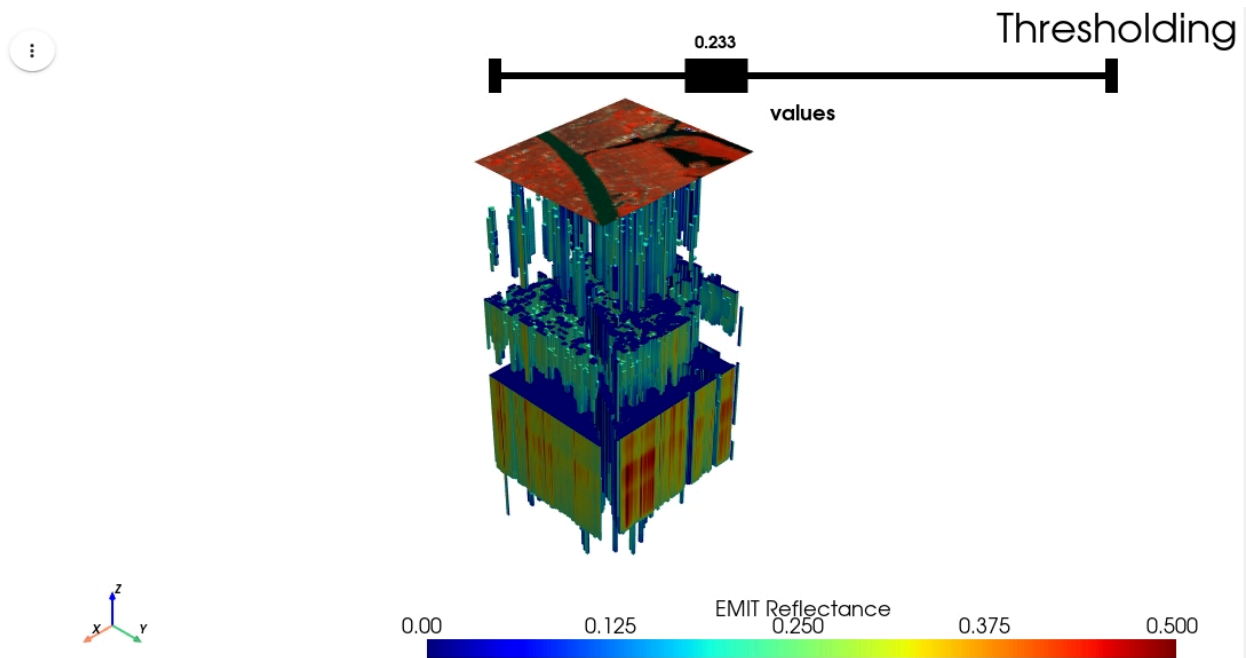


Figure 122: The interactive thresholding widget for the EMIT dataset.

The thresholding widget allows you to interactively set reflectance cutoff values, revealing only pixels meeting your criteria. This technique is particularly useful for preliminary material classification or identifying areas of interest for detailed analysis. Drag the slider to adjust the threshold value and see corresponding pixels with reflectance values above the threshold.

24.11. Key Takeaways

This chapter introduced fundamental concepts and techniques for working with hyperspectral data using the HyperCoast package:

Hyperspectral Data Structure: Hyperspectral data is three-dimensional with two spatial dimensions and one spectral dimension, enabling more sophisticated analysis than traditional multispectral imagery. The continuous spectral coverage from instruments like EMIT allows detailed material discrimination and quantitative surface property analysis.

Data Discovery and Access: The workflow begins with data discovery and acquisition through both programmatic and interactive search methods. Authentication through NASA Earthdata services provides access to extensive archives of high-quality hyperspectral datasets.

Visualization Techniques: False-color composites reveal features invisible to the human eye, while spectral profiling enables quantitative material property analysis. Three-dimensional visualization through image cubes provides intuitive understanding of spatial-spectral relationships.

Interactive Exploration: Slicing and thresholding widgets facilitate data exploration and analysis, helping identify optimal wavelength ranges for specific applications and enabling preliminary material classification based on spectral characteristics.

These fundamental techniques provide the foundation for advanced hyperspectral analysis methods, including automated classification, spectral unmixing, and quantitative parameter retrieval.

24.12. Exercises

Practice your understanding of hyperspectral data visualization with these hands-on exercises:

24.12.1. Exercise 1: Data Discovery and Acquisition

Choose a geographic region of interest (such as your local area or a study site) and search for available EMIT data using both programmatic and interactive methods. Compare the results from both approaches and identify the most suitable dataset for your area. Download at least one dataset and examine its metadata and characteristics.

24.12.2. Exercise 2: Spectral Profile Analysis

Using the downloaded dataset, extract spectral profiles from different land cover types (e.g., vegetation, water, urban areas, bare soil). Create plots showing the spectral signatures of these different materials and identify key wavelength regions where they can be distinguished from each other.

24.12.3. Exercise 3: Band Combination Exploration

Experiment with different band combinations for creating false-color composites. Try combinations that highlight vegetation (using near-infrared bands), water features (using shortwave infrared bands), and geological features. Document which combinations work best for different types of features and explain why.

24.12.4. Exercise 4: Interactive Visualization

Create an interactive map visualization of your hyperspectral data with multiple band combinations. Add spectral profiling capabilities and extract profiles from at least five different locations. Export the spectral profiles and create comparative plots showing how different materials exhibit distinct spectral characteristics.

24.12.5. Exercise 5: 3D Cube Analysis

Generate 3D visualizations of a subset of your hyperspectral data. Use the interactive slicing tool to explore how different wavelengths reveal different surface features. Identify wavelength ranges that are most sensitive to specific materials in your scene and document your findings with screenshots and explanations.

Chapter 25. High-Performance Geospatial Analytics with DuckDB

25.1. Introduction

DuckDB¹²⁶ is a revolutionary analytical database engine that transforms how we work with geospatial data. In today's data-driven world, geospatial datasets are growing exponentially in both size and complexity—from GPS tracking data and satellite imagery to urban planning datasets and environmental monitoring systems. Traditional data processing tools often struggle with these demands, leading to slow analysis, complex setup requirements, and limitations in handling diverse data formats.

DuckDB addresses these challenges by bringing high-performance, SQL-based analysis capabilities directly to your geospatial workflows. Think of it as having the power of a sophisticated database engine embedded right in your Python environment, without the overhead of managing database servers or complex infrastructure.

25.1.1. What Makes DuckDB Special for Geospatial Work?

DuckDB is designed from the ground up for analytical workloads (Online Analytical Processing - OLAP), which makes it exceptionally well-suited for geospatial analysis. Here's why this matters:

Analytical vs. Transactional Focus: Unlike traditional databases designed for transactional workloads (like recording sales or user interactions), DuckDB excels at the kind of operations geospatial analysts perform daily: aggregations across large datasets, complex spatial joins, and analytical queries that process millions of geographic features.

In-Process Architecture: DuckDB runs as an embedded database directly within your Python process. This means no database servers to install, configure, or maintain—just import the library and start analyzing. This is particularly valuable for geospatial work where you often need to quickly explore datasets or share analysis code with colleagues.

Columnar Storage: DuckDB stores data in a column-oriented format, which dramatically speeds up analytical queries. When you're calculating the average population of cities or finding all points within a certain distance, you only need to read the relevant columns, not entire rows.

Direct Format Support: Perhaps most importantly for geospatial analysts, DuckDB can directly query multiple data formats without requiring traditional Extract-Transform-Load (ETL) processes. You can analyze GeoJSON files, Shapefiles, Parquet files, and CSV data directly without first importing them into a database.

25.2. Learning Objectives

By the end of this chapter, you will be able to:

- Understand the architecture and advantages of DuckDB for geospatial analysis
- Install and configure DuckDB with spatial capabilities
- Use SQL basics for data querying and manipulation
- Integrate DuckDB with Python for analytical workflows
- Load and export various geospatial data formats

¹²⁶<https://duckdb.org>

- Work with geometry data types and spatial functions
- Perform spatial relationships analysis
- Execute high-performance spatial joins and operations
- Visualize geospatial data with Python libraries
- Conduct large-scale geospatial data analysis

25.3. Installation and Setup

25.3.1. Installing Required Packages

DuckDB can be installed easily using pip, Python's package manager. We'll also install some helpful libraries for working with geospatial data:

```
# %pip install duckdb pygis
```

The `duckdb` package provides the core DuckDB database engine with Python bindings, while `leafmap` and `pandas` offer additional utilities for working with geospatial data and data analysis in Python.

25.3.2. Library Import and Configuration

Before we start working with spatial data, we need to import the necessary libraries and configure our Jupyter environment for optimal display of results.

```
# Configure jupysql for optimal output
%config SqlMagic.autopandas = True
%config SqlMagic.feedback = False
%config SqlMagic.displaycon = False
```

These configuration settings optimize our Jupyter environment for spatial analysis. The `autopandas = True` setting automatically converts SQL query results to pandas DataFrames for easy manipulation. Setting `feedback = False` reduces verbose output to keep notebooks clean, while `displaycon = False` hides connection details from the output.

```
import duckdb
import leafmap
import pandas as pd

# Import jupysql Jupyter extension for SQL magic commands
%load_ext sql
```

25.3.3. Understanding DuckDB Connections

DuckDB offers flexible connection options depending on your needs:

```
# Connect to in-memory database
con = duckdb.connect()
```

```
# Connect to persistent file-based database
con = duckdb.connect('filename.db')

# Connect using SQL magic for jupyter notebooks
%sql duckdb:///memory:
```

Connection Types Explained:

In-Memory Database (`duckdb.connect()`): Data exists only while your program runs, making it perfect for exploratory data analysis, temporary calculations, and processing data that doesn't need to persist between sessions.

File-Based Database (`duckdb.connect('filename.db')`): Data is saved to disk and persists between sessions, making it ideal for building data warehouses, storing processed results, and sharing databases with colleagues.

SQL Magic Connection (`%sql duckdb:///memory:`): Enables using `%%sql` cells in Jupyter notebooks for more convenient SQL writing.

25.3.4. Installing and Loading Extensions

DuckDB uses a modular extension system to provide specialized functionality. This keeps the core engine lightweight while allowing you to add capabilities as needed:

```
# Install and load essential extensions
con.install_extension("httpfs") # For remote file access
con.load_extension("httpfs")

con.install_extension("spatial") # For spatial operations
con.load_extension("spatial")
```

Essential Extensions for Geospatial Work:

HTTPFS Extension: Enables reading data directly from web URLs and cloud storage. This is crucial for geospatial work because many spatial datasets are hosted online, allowing you to analyze data without downloading large files first. The extension supports reading from AWS S3, Google Cloud Storage, and HTTP URLs.

Spatial Extension: Provides comprehensive geospatial functionality including geometry data types (Point, LineString, Polygon), spatial functions (distance, intersection, buffer operations), support for reading and writing spatial file formats, and spatial indexing for performance optimization.

You can check the list of extensions with the following command:

```
con.sql("FROM duckdb_extensions();")
```

DuckDB only shows the first 10 rows and the last 10 rows of a table. To show all rows, you can convert the table to a pandas DataFrame using the `.df()` method:

```
con.sql("FROM duckdb_extensions();").df()
```

	extension_name	loaded	installed	install_path	description	aliases	extension_version	install_mode	installed_from
0	arrow	False	False		A zero-copy data integration between Apache Ar...	[]		None	
1	autocomplete	False	False		Adds support for autocomplete in the shell	[]		None	
2	aws	False	False		Provides features that depend on the AWS SDK	[]		None	
3	azure	False	False		Adds a filesystem abstraction for Azure blob s...	[]		None	
4	core_functions	True	True	(BUILT-IN)	Core function library	[]		STATICALLY_LINKED	
5	delta	False	False		Adds support for Delta Lake	[]		None	
6	excel	False	False		Adds support for Excel-like format strings	[]		None	
7	fts	False	False		Adds support for Full-Text Search Indexes	[]		None	
8	https	True	True	/home/qiusheng/duckdb/extensions/v1.2.2/linux...	Adds support for reading and writing files ove...	[http, https, s3]	c225324	REPOSITORY	core
9	iceberg	False	False		Adds support for Apache Iceberg	[]		None	
10	icu	True	True	(BUILT-IN)	Adds support for time zones and collations usi...	[]		STATICALLY_LINKED	
11	inet	False	False		Adds support for IP-related data types and fun...	[]		None	
12	jemalloc	True	True	(BUILT-IN)	Overwrites system allocator with JEMalloc	[]		STATICALLY_LINKED	
13	json	True	True	(BUILT-IN)	Adds support for JSON operations	[]		STATICALLY_LINKED	
14	motherduck	False	False		Enables motherduck integration with the system	[md]		None	
15	mysql_scanner	False	False		Adds support for connecting to a MySQL database	[mysql]		None	
16	parquet	True	True	(BUILT-IN)	Adds support for reading and writing parquet f...	[]		STATICALLY_LINKED	
17	postgres_scanner	False	False		Adds support for connecting to a Postgres data...	[postgres]		None	
18	spatial	True	True	/home/qiusheng/duckdb/extensions/v1.2.2/linux...	Geospatial extension that adds support for wor...	[]	3bb37f8	REPOSITORY	core
19	sqlite_scanner	False	False		Adds support for reading and writing SQLite da...	[sqlite, sqlite3]		None	
20	tpcds	False	False		Adds TPC-DS data generation and query support	[]		None	
21	tpch	True	True	(BUILT-IN)	Adds TPC-H data generation and query support	[]		STATICALLY_LINKED	
22	ui	False	False		Adds local UI for DuckDB	[]		None	
23	vss	False	False		Adds indexing support to accelerate Vector Sim...	[]		None	

25.4. SQL Basics for Spatial Analysis

25.4.1. Understanding SQL for Geospatial Work

SQL (Structured Query Language) is the standard language for working with databases, and it's particularly powerful for geospatial analysis. If you're new to SQL, think of it as a way to ask questions about your data using English-like commands. For example, instead of writing complex loops in Python, you can simply ask: "SELECT all cities WHERE population is greater than 1 million."

25.4.2. Sample Datasets

We'll use two sample datasets to learn fundamental concepts. The [Cities Dataset](https://opengeos.org/data/duckdb/cities.csv)¹²⁷ contains information about world cities including their locations, populations, and countries, while the [Countries Dataset](https://opengeos.org/data/duckdb/countries.csv)¹²⁸ contains country-level information including boundaries and attributes.

These datasets are perfect for learning because they represent common geospatial data types you'll encounter in real-world projects.

25.4.3. Reading Data Directly from URLs

One of DuckDB's most powerful features is its ability to read data directly from web URLs without downloading files first. This is especially useful for geospatial data, which is often large and frequently updated.

¹²⁷<https://opengeos.org/data/duckdb/cities.csv>

¹²⁸<https://opengeos.org/data/duckdb/countries.csv>


```
%%sql
SELECT * FROM 'https://opengeos.org/data/duckdb/cities.csv';
```

	id	name	country	latitude	longitude	population
0	1	Bombo	UGA	0.58330	32.53330	75000
1	2	Fort Portal	UGA	0.67100	30.27500	42670
2	3	Potenza	ITA	40.64200	15.79900	69060
3	4	Campobasso	ITA	41.56300	14.65600	50762
4	5	Aosta	ITA	45.73700	7.31500	34062
...	...	...	...	...	...	...
1244	1245	Rio de Janeiro	BRA	-22.92502	-43.22502	11748000
1245	1246	Sao Paulo	BRA	-23.55868	-46.62502	18845000
1246	1247	Sydney	AUS	-33.92001	151.18518	4630000
1247	1248	Singapore	SGP	1.29303	103.85582	5183700
1248	1249	Hong Kong	CHN	22.30498	114.18501	7206000

1249 rows × 6 columns

In this query, `SELECT *` means “show me all columns,” while `FROM 'https://...'` tells DuckDB to read data directly from a web URL. The `%%sql` magic command allows us to write SQL in a Jupyter notebook cell.

```
%%sql
SELECT * FROM 'https://opengeos.org/data/duckdb/countries.csv';
```

	id	Country	Alpha2_code	Alpha3_code	Numeric_code	Latitude	Longitude
0	1	Afghanistan	AF	AFG	4	33.0000	65.0
1	2	Albania	AL	ALB	8	41.0000	20.0
2	3	Algeria	DZ	DZA	12	28.0000	3.0
3	4	American Samoa	AS	ASM	16	-14.3333	-170.0
4	5	Andorra	AD	AND	20	42.5000	1.6
...	...	...	...	...	...	...	...
238	239	Wallis and Futuna	WF	WLF	876	-13.3000	-176.2
239	240	Western Sahara	EH	ESH	732	24.5000	-13.0
240	241	Yemen	YE	YEM	887	15.0000	48.0
241	242	Zambia	ZM	ZMB	894	-15.0000	30.0
242	243	Zimbabwe	ZW	ZWE	716	-20.0000	30.0

243 rows × 7 columns

25.4.4. Creating Persistent Tables

While reading data directly from URLs is convenient for exploration, creating tables gives us several advantages including faster subsequent queries (data is cached locally), the ability to index the data for better performance, and more consistent column names and data types.

```
%%sql
CREATE OR REPLACE TABLE cities AS SELECT * FROM 'https://opengeos.org/data/
duckdb/cities.csv';
```

In this command, `CREATE OR REPLACE TABLE` creates a new table (or replaces if it exists), `cities` is our chosen table name, and `AS SELECT *` means “populate this table with all data from the source.”

Let’s also create a table for the countries dataset:

```
%%sql
CREATE OR REPLACE TABLE countries AS SELECT * FROM 'https://opengeos.org/data/
duckdb/countries.csv';
```

25.4.5. Viewing Your Data

Now that we have tables, let’s examine their contents. DuckDB supports a simplified syntax that’s particularly convenient for data exploration:

```
%%sql
FROM cities;
```

Note: `FROM cities` is shorthand for `SELECT * FROM cities`. This simplified syntax makes data exploration faster.

```
%%sql
FROM countries;
```

25.4.6. Essential SQL Commands

SQL commands are the building blocks of data analysis. Let’s explore the most important ones for geospatial work, starting with basic data selection and moving toward more complex operations.

25.4.6.1. Selecting Data

The `SELECT` statement is your primary tool for examining data. Think of it as asking: “Show me specific information from my dataset.”

```
%%sql
SELECT * FROM cities LIMIT 10;
```

Breaking this down: `SELECT *` means “show all columns,” `FROM cities` specifies which table to query, and `LIMIT 10` restricts output to first 10 rows (useful for large datasets).

25.4.6.2. Choosing Specific Columns

Often you don’t need all columns - just select what’s relevant to your analysis:

```
%%sql
SELECT name, country FROM cities LIMIT 10;
```

This approach is important for geospatial work because it reduces data transfer and memory usage, focuses your analysis on relevant attributes, and makes results easier to read and understand.

25.4.6.3. Finding Unique Values

Understanding the distinct values in your dataset is crucial for spatial analysis:

```
%%sql
SELECT DISTINCT country FROM cities LIMIT 10;
```

Common use cases include determining how many different countries are represented, what are the unique administrative regions, and which categories exist in a land-use classification.

25.4.6.4. Counting and Aggregation

Aggregation functions help you understand your data’s characteristics:

```
%%sql
SELECT COUNT(*) FROM cities;
```

What this tells us: The total number of cities in our dataset.

```
%%sql
SELECT COUNT(DISTINCT country) FROM cities;
```

What this tells us: How many different countries are represented.

```
%%sql
SELECT MAX(population) FROM cities;
```

What this tells us: The population of the largest city.

```
%%sql
SELECT MIN(population) FROM cities;
```

What this tells us: The population of the smallest city.

```
%%sql
SELECT AVG(population) FROM cities;
```

What this tells us: The average city population across all cities.

Aggregations are important in geospatial analysis because they help you understand the scale and distribution of your data, identify outliers that might need special attention, calculate summary statistics for regions or categories, and validate data quality before spatial operations.

25.4.7. Filtering and Sorting Data

Real-world geospatial analysis often requires focusing on specific subsets of data or examining data in a particular order. SQL provides powerful tools for both filtering and sorting.

25.4.7.1. Filtering with WHERE Clauses

The `WHERE` clause lets you specify conditions to filter your data. Think of it as asking: “Show me only the records that meet these criteria.”

```
%%sql
SELECT * FROM cities WHERE population > 1000000 LIMIT 10;
```

Understanding this query: `WHERE population > 1000000` filters for cities with more than 1 million people, which is equivalent to finding “major cities” or “megacities.” Common spatial filters might include elevation ranges, area thresholds, or specific regions.

25.4.7.2. Sorting Your Results

Sorting helps you understand patterns in your data and identify extremes:

```
%%sql
SELECT * FROM cities ORDER BY population DESC LIMIT 10;
```

This query shows us several important concepts: `ORDER BY population DESC` sorts by population in descending order (largest first), `LIMIT 10` gives us the top 10 most populous cities, and this represents a common pattern in geospatial analysis for finding the largest, smallest, northernmost, etc.

25.4.7.3. Grouping and Aggregating Data

Grouping is essential for understanding spatial patterns and regional characteristics:

```
%%sql
SELECT country, COUNT(*) as city_count, AVG(population) as avg_population
FROM cities
GROUP BY country
ORDER BY city_count DESC
LIMIT 10;
```

Breaking down this complex query: `GROUP BY country` creates groups for each unique country, `COUNT(*) as city_count` counts how many cities are in each country, `AVG(population) as avg_population` calculates the average city population per country, and `ORDER BY city_count DESC` shows countries with the most cities first.

Grouping is important in geospatial analysis for understanding regional patterns (population distribution, economic activity), calculating statistics by administrative boundaries, comparing characteristics across different areas, and preparing data for choropleth maps and regional visualizations.

25.5. Python API Integration

DuckDB's Python integration is one of its strongest features, allowing you to seamlessly combine SQL's power with Python's rich ecosystem of data science libraries. This integration is particularly valuable for geospatial work, where you might need to combine database operations with mapping, visualization, or advanced analytics.

25.5.1. Understanding Result Formats

DuckDB can return query results in different formats, each optimized for different use cases:

```
# Execute SQL and get different result formats
con.sql('SELECT 42').fetchall()      # Python objects
```

The `fetchall()` method returns results as Python lists and tuples, making it best for small result sets, when you need basic Python data structures, and for simple data processing without pandas overhead.

```
con.sql('SELECT 42').df()            # Pandas DataFrame
```

The `df()` method returns results as pandas DataFrames, making it best for data analysis and manipulation, integration with other pandas-based workflows, and when you need pandas' rich functionality (groupby, pivot, etc.).

```
con.sql('SELECT 42').fetchnumpy()    # NumPy Arrays
```

The `fetchnumpy()` method returns results as NumPy arrays, making it best for high-performance numerical computations, machine learning workflows, and when working with large numerical datasets.

25.5.2. Seamless DataFrame Integration

One of DuckDB's most powerful features is its ability to query pandas DataFrames directly, treating them as if they were database tables:

```
# Query Pandas DataFrames directly
pandas_df = pd.DataFrame({'a': [42]})
con.sql('SELECT * FROM pandas_df')
```

This capability is revolutionary because there's no need to import data into a database first, you can combine SQL's power with Python's data processing, and you can mix DataFrame operations with SQL operations seamlessly.

```
# Convert remote data to DataFrame
df = con.read_csv('https://opengeos.org/data/duckdb/cities.csv').df()
df.head()
```

Practical applications for geospatial work include loading spatial data from various sources into DataFrames, using SQL for complex spatial queries, using pandas for data cleaning and preprocessing, and combining both approaches in a single workflow.

25.5.3. Result Conversion and Export

The `con.sql()` method returns a `QueryResult` object, which can be converted to various formats using the `write_csv()`, `write_parquet()`, and `write_geojson()` methods.

```
# Write results to various formats
con.sql('SELECT 42').write_parquet('out.parquet')
con.sql('SELECT 42').write_csv('out.csv')
con.sql("COPY (SELECT 42) TO 'out.parquet'")
```

25.5.4. Persistent Storage

The `duckdb.connect()` function can be used to create a persistent connection to a DuckDB database. This is useful for storing data in a file and querying it later.

```
# Create persistent database connection
# create a connection to a file called 'file.db'
con_persistent = duckdb.connect('file.db')
# create a table and load data into it
con_persistent.sql(
    'CREATE TABLE IF NOT EXISTS cities AS FROM read_csv_auto("https://opengeos.
org/data/duckdb/cities.csv")'
)
# query the table
con_persistent.table('cities').show()
```

Once you are done with the analysis, you can close the connection to the database by calling `close` method on the connection object.

```
# explicitly close the connection
con_persistent.close()
```

You can also use a context manager to create a connection to a DuckDB database. This is useful for creating a connection to a database and then closing it automatically when you're done.

```
with duckdb.connect('file.db') as con:
    con.sql(
        'CREATE TABLE IF NOT EXISTS cities AS FROM read_csv_auto("https://
opengeos.org/data/duckdb/cities.csv")'
    )
    con.table('cities').show(max_rows=5)
# the context manager closes the connection automatically
```

25.6. Data Import

One of DuckDB’s greatest strengths is its ability to work with data in multiple formats without requiring traditional Extract-Transform-Load (ETL) processes. This “zero-ETL” approach means you can analyze data directly from its source format, saving time and storage space.

25.6.1. Understanding Data Formats in Geospatial Work

Geospatial data comes in many formats, each with different strengths. CSV files are simple, human-readable, and widely supported, but have limited spatial capabilities. JSON/GeoJSON formats are flexible, support complex geometries, and are web-friendly. Parquet files are highly compressed, fast to read, and excellent for large datasets. Shapefiles represent the traditional GIS format that’s widely supported but has some limitations. GeoPackage is a modern spatial database format that supports multiple layers.

25.6.2. Downloading Sample Data

Let’s start by downloading sample data to work with:

```
url = "https://opengeos.org/data/duckdb/cities.zip"
leafmap.download_file(url, unzip=True)
```

25.6.3. Working with CSV Files

CSV (Comma-Separated Values) files are the most common format for tabular data. While they don’t natively support spatial geometries, they often contain coordinate information.

```
# Read CSV with auto-detection
con = duckdb.connect()
con.read_csv('cities.csv')
```

Auto-detection provides several benefits: it automatically determines column types (integer, float, text), detects delimiters and quote characters, and handles common CSV variations without manual configuration.

```
# Read CSV with specific options
con.read_csv('cities.csv', header=True, sep=',')
```

You should specify options when auto-detection fails, for better performance with very large files, or when you need specific data types.

```
# Use parallel CSV reader
con.read_csv('cities.csv', parallel=True)
```

Parallel processing offers significant advantages by being dramatically faster for large files and utilizing multiple CPU cores, which is especially beneficial for files with millions of rows

```
# Read CSV directly in SQL
con.sql("SELECT * FROM 'cities.csv'")
```

Direct SQL reading benefits:

- No need to explicitly create a connection first
- Perfect for one-off queries
- Keeps your workflow streamlined

```
# Call read_csv from within SQL
con.sql("SELECT * FROM read_csv_auto('cities.csv')")
```

Using functions in SQL benefits:

- More explicit control over reading parameters
- Useful when you need to specify options within SQL queries
- Maintains consistency across different data sources

25.6.4. JSON Files

DuckDB supports reading JSON files directly from local files or URLs.

```
# Read JSON files
con.read_json('cities.json')
```

```
# Read JSON directly in SQL
con.sql("SELECT * FROM 'cities.json'")
```

```
# Call read_json from within SQL
con.sql("SELECT * FROM read_json_auto('cities.json')")
```

25.6.5. DataFrames

Pandas DataFrames stored in local variables can be queried as if they are regular tables within DuckDB.


```
df = pd.read_csv('cities.csv')
df
```

```
con.sql('SELECT * FROM df').fetchall()
```

25.6.6. Parquet Files

Parquet is a columnar format that is efficient for storing and querying large datasets. DuckDB has native support for Parquet files.

```
# Read from a single Parquet file
con.read_parquet('cities.parquet')
```

```
# Read Parquet directly in SQL
con.sql("SELECT * FROM 'cities.parquet'")
```

```
# Call read_parquet from within SQL
con.sql("SELECT * FROM read_parquet('cities.parquet')")
```

25.6.7. Spatial Data Formats

In this section, we will learn how to read spatial data from various formats into DuckDB using the [spatial extension](https://duckdb.org/docs/stable/core_extensions/spatial/overview.html)¹²⁹.

To read spatial data, we first need to load the spatial extension.

```
con.load_extension('spatial')
```

Once the spatial extension is loaded, we can check the list of supported spatial file formats with the following command:

```
con.sql('SELECT * FROM ST_Drivers()')
```

¹²⁹https://duckdb.org/docs/stable/core_extensions/spatial/overview.html

short_name varchar	long_name varchar	can_create boolean	can_copy boolean	can_open boolean	help_url varchar
ESRI Shapefile	ESRI Shapefile	true	false	true	https://gdal.org/drivers/vector/shapefile.html
MapInfo File	MapInfo File	true	false	true	https://gdal.org/drivers/vector/mitab.html
UK .NTF	UK .NTF	false	false	true	https://gdal.org/drivers/vector/ntf.html
LV BAG	Kadaster LV BAG Extract 2.0	false	false	true	https://gdal.org/drivers/vector/lvbag.html
S57	IHO S-57 (ENC)	true	false	true	https://gdal.org/drivers/vector/s57.html
DGN	Microstation DGN	true	false	true	https://gdal.org/drivers/vector/dgn.html
OGR_VRT	VRT - Virtual Datasource	false	false	true	https://gdal.org/drivers/vector/vrt.html
Memory	Memory	true	false	true	NULL
CSV	Comma Separated Value (.csv)	true	false	true	https://gdal.org/drivers/vector/csv.html
GML	Geography Markup Language (GML)	true	false	true	https://gdal.org/drivers/vector/gml.html
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
VDV	VDV-451/VDV-452/INTREST Data Format	true	false	true	https://gdal.org/drivers/vector/vdv.html
MVT	Mapbox Vector Tiles	true	false	true	https://gdal.org/drivers/vector/mvt.html
NGW	NextGIS Web	true	true	true	https://gdal.org/drivers/vector/ngw.html
MapML	MapML	true	false	true	https://gdal.org/drivers/vector/mapml.html
GTFS	General Transit Feed Specification	false	false	true	https://gdal.org/drivers/vector/gtfs.html
PMTiles	ProtoMap Tiles	true	false	true	https://gdal.org/drivers/vector/pmtiles.html
JSONFG	OGC Features and Geometries JSON	true	false	true	https://gdal.org/drivers/vector/jsonfg.html
TIGER	U.S. Census TIGER/Line	false	false	true	https://gdal.org/drivers/vector/tiger.html
AVCBin	Arc/Info Binary Coverage	false	false	true	https://gdal.org/drivers/vector/avcbin.html
AVCE00	Arc/Info E00 (ASCII) Coverage	false	false	true	https://gdal.org/drivers/vector/avce00.html
54 rows (20 shown)					6 columns

With the spatial extension loaded, we can read spatial data from various formats into DuckDB using the `ST_Read` function.

First, let's read a GeoJSON file:

```
# Read GeoJSON
con.sql("SELECT * FROM ST_Read('cities.geojson')")
```

Note that the `geom` column is a `WKB` (Well-Known Binary) column, which is a binary representation of the geometry. DuckDB can read this column as a geometry object.

To create a table from a GeoJSON file, use the `CREATE TABLE` statement with the `ST_Read` function.

```
# Create spatial table from GeoJSON
con.sql("CREATE TABLE cities AS SELECT * FROM ST_Read('cities.geojson')")
```

Show the first and last 10 rows of the `cities` table:

```
con.table('cities')
```

The `ST_Read` function can read data any vector format supported by GDAL. For example, we can read an ESRI Shapefile into DuckDB:

```
con.sql("SELECT * FROM ST_Read('cities.shp')")
```

Create a table from the Shapefile:

```
con.sql(
    """
    CREATE TABLE IF NOT EXISTS cities2 AS
    SELECT * FROM ST_Read('cities.shp')
    """
)
```

```
con.table('cities2')
```

25.7. Data Export

DuckDB can export data to various formats, such as CSV, Parquet, GeoJSON, Shapefile, and more. In this section, we will learn how to export spatial data from DuckDB to various formats.

25.7.1. Sample Spatial Data Setup

Let's create a table from a sample spatial dataset:

```
con = duckdb.connect()
con.load_extension('spatial')
con.sql("""
CREATE TABLE IF NOT EXISTS cities AS
FROM 'https://opengeos.org/data/duckdb/cities.parquet'
""")
```

```
con.table("cities").show()
```

25.7.2. Export to DataFrames

```
con.table("cities").df()
```

25.7.3. Export to Files

Export data to files using the `COPY` statement:

```
# Export to CSV
con.sql("COPY cities TO 'cities.csv' (HEADER, DELIMITER ',')")
```

```
con.sql(
    "COPY (SELECT * FROM cities WHERE country='USA') TO 'cities_us.csv' (HEADER,
```

```
DELIMITER ',')"  
)
```

25.7.4. Export to JSON

Similarly, we can export data to JSON using the `COPY` statement:

```
con.sql("COPY cities TO 'cities.json'")
```

```
con.sql("COPY (SELECT * FROM cities WHERE country='USA') TO 'cities_us.json'")
```

25.7.5. Export to Excel

To export data to Excel, we can use the `COPY` statement with the `FORMAT GDAL` and `DRIVER 'XLSX'` options.

```
con.sql(  
    "COPY (SELECT * EXCLUDE geometry FROM cities) TO 'cities.xlsx' WITH (FORMAT  
    GDAL, DRIVER 'XLSX')"  
)
```

25.7.6. Export to Parquet

DuckDB can export data to Parquet files using the `COPY` statement with the `FORMAT PARQUET` option. No GDAL driver is needed.

```
con.sql("COPY cities TO 'cities.parquet' (FORMAT PARQUET)")
```

```
con.sql(  
    "COPY (SELECT * FROM cities WHERE country='USA') TO  
    'cities_us.parquet' (FORMAT PARQUET)"  
)
```

25.7.7. Export Spatial Formats

The examples below show how to export spatial data to various spatial formats, including GeoJSON, Shapefile, and GeoPackage.

```
# Export to GeoJSON  
con.sql("COPY cities TO 'cities.geojson' WITH (FORMAT GDAL, DRIVER 'GeoJSON')")
```

```
con.sql(
  "COPY (SELECT * FROM cities WHERE country='USA') TO 'cities_us.geojson' WITH
  (FORMAT GDAL, DRIVER 'GeoJSON')"
)
```

```
# Export to Shapefile
con.sql("COPY cities TO 'cities.shp' WITH (FORMAT GDAL, DRIVER 'ESRI
Shapefile')")
```

```
# Export to GeoPackage
con.sql("COPY cities TO 'cities.gpkg' WITH (FORMAT GDAL, DRIVER 'GPKG')")
```

25.8. Working with Geometries

Geometries are the fundamental building blocks of spatial data. Understanding how to create, manipulate, and analyze geometries is essential for any geospatial workflow. DuckDB's spatial extension provides comprehensive support for geometry types and operations, following the Open Geospatial Consortium (OGC) standards.

25.8.1. Understanding Spatial Data Types

Before diving into examples, let's understand the basic geometry types you'll encounter. A Point represents a single location (e.g., a city, a GPS coordinate), while a LineString represents a path or route (e.g., a road, a river). Polygons represent areas with boundaries (e.g., a country, a lake). More complex geometries include MultiPoint, MultiLineString, and MultiPolygon, which are collections of the basic types, and GeometryCollection, which can contain mixed geometry types.

25.8.2. Sample Data Setup

Let's start with a real-world spatial database containing New York City data:

```
# Download NYC sample data
url = "https://opengeos.org/data/duckdb/nyc_data.db.zip"
leafmap.download_file(url, unzip=True)
```

```
# Connect to spatial database
con = duckdb.connect("nyc_data.db")
con.install_extension("spatial")
con.load_extension("spatial")
```

```
con.sql("SHOW TABLES;")
```

25.8.3. Creating and Understanding Geometries

Let's create examples of each geometry type to understand how they work:

```
con.sql("""
CREATE or REPLACE TABLE samples (name VARCHAR, geom GEOMETRY);

INSERT INTO samples VALUES
  ('Point', ST_GeomFromText('POINT(-100 40)')),
  ('Linestring', ST_GeomFromText('LINESTRING(0 0, 1 1, 2 1, 2 2)')),
  ('Polygon', ST_GeomFromText('POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))')),
  ('PolygonWithHole', ST_GeomFromText('POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),(1
1, 1 2, 2 2, 2 1, 1 1))')),
  ('Collection', ST_GeomFromText('GEOMETRYCOLLECTION(POINT(2 0),POLYGON((0 0, 1
0, 1 1, 0 1, 0 0))'))));

SELECT * FROM samples;

""")
```

Understanding [Well-Known Text \(WKT\)](#)¹³⁰ is essential for working with geometries. The `ST_GeomFromText()` function creates geometries from WKT format, which is a standard text representation for geometries. Coordinates are typically in (X, Y) or (longitude, latitude) order, and the spatial reference system affects how these coordinates are interpreted.

When printing a geometry, we can use the `ST_AsText()` function to get the WKT representation of the geometry.

```
con.sql("SELECT name, ST_AsText(geom) AS geometry FROM samples;")
```

To save the geometry to a file, we can use `COPY` statement with the `FORMAT GDAL` and `DRIVER 'GeoJSON'` options.

```
con.sql("""
COPY samples TO 'samples.geojson' (FORMAT GDAL, DRIVER GeoJSON);
""")
```

25.8.4. Working with Points

Points represent single locations on Earth with coordinates:

```
con.sql("""
SELECT ST_AsText(geom)
FROM samples
```

¹³⁰<https://tinyurl.com/y3m3aeod>

```
WHERE name = 'Point';
""")
```

Extract the X and Y coordinates from the geometry:

```
con.sql("""
SELECT ST_X(geom), ST_Y(geom)
FROM samples
WHERE name = 'Point';
""")
```

Read the `nyc_subway_stations` table from the database:

```
con.sql("""
SELECT * FROM nyc_subway_stations
""")
```

Select the `name` and `geom` columns from the `nyc_subway_stations` table:

```
con.sql("""
SELECT name, ST_AsText(geom)
FROM nyc_subway_stations
LIMIT 10;
""")
```

25.8.5. Working with LineStrings

LineStrings represent paths between locations:

```
con.sql("""
SELECT ST_AsText(geom)
FROM samples
WHERE name = 'Linestring';
""")
```

Calculate the length of the line using the `ST_Length` function:

```
con.sql("""
SELECT ST_Length(geom)
FROM samples
WHERE name = 'Linestring';
""")
```

25.8.6. Working with Polygons

Polygons represent areas with boundaries:

```
con.sql("""
SELECT ST_AsText(geom)
FROM samples
WHERE name = 'Polygon';
""")
```

Calculate the area and perimeter of the polygon using the `ST_Area()` and `ST_Perimeter()` functions:

```
con.sql("""
SELECT
    ST_Area(geom) as area,
    ST_Perimeter(geom) as perimeter
FROM samples
WHERE name = 'Polygon';
""")
```

25.9. Spatial Relationships

Spatial relationships are the foundation of geographic analysis—they help us understand how features in space are connected, positioned, or interact with one another. These relationships allow us to answer critical questions like “Which neighborhoods contain subway stations?”, “How far is this park from the nearest hospital?”, or “What areas are affected by a floodplain?”

Think of spatial relationships as enabling us to ask geography-aware questions. For example:

- **Proximity:** “Which cities are within 100 km of the coast?”
- **Containment:** “Does this district fall inside a flood zone?”
- **Intersection:** “Does this road pass through a protected forest?”

25.9.1. Understanding Spatial Predicates

Spatial predicates are logical functions that evaluate the relationship between two geometries and return a boolean value (true or false). These form the basis of spatial queries in SQL and GIS analysis.

Common spatial predicates include:

- **ST_Equals(a, b)**: Checks if two geometries are exactly the same in structure and location.
- **ST_Intersects(a, b)**: Returns `TRUE` if the geometries share any spatial overlap.
- **ST_Contains(a, b)**: Tests if geometry `a` completely contains geometry `b`.
- **ST_Within(a, b)**: The reverse of `ST_Contains`; `a` lies entirely within `b`.
- **ST_Touches(a, b)**: The geometries share a border but do not overlap.
- **ST_Disjoint(a, b)**: The geometries have no points in common.

Understanding these relationships is essential for conducting spatial joins, containment tests, proximity searches, and more.

25.9.2. Working with Real Spatial Data

Let's bring these concepts to life using real data from New York City, specifically subway stations and neighborhood boundaries.

First, let's find a subway station by name:

```
# Find subway station
con.sql("""
SELECT name, geom
FROM nyc_subway_stations
WHERE name = 'Broad St';
""")
```

25.9.3. Testing Spatial Equality

To check if a geometry exactly matches another, use `ST_Equals()`:

```
# Test spatial equality
con.sql("""
SELECT name
FROM nyc_subway_stations
WHERE ST_Equals(geom, ST_GeomFromText('POINT (583571.9059213118
4506714.341192182)'));
""")
```

This query finds stations whose geometry precisely matches the given point—useful for quality control or deduplication.

25.9.4. Point-in-Polygon Analysis

A classic use case: determine which polygon (e.g., neighborhood or district) contains a given point (e.g., subway station).

```
con.sql("FROM nyc_neighborhoods LIMIT 5")
```

```
# Find neighborhood containing a point
con.sql("""
SELECT name, boroname
FROM nyc_neighborhoods
WHERE ST_Intersects(geom, ST_GeomFromText('POINT(583571 4506714)'));
""")
```

25.9.5. Proximity Analysis with Distance

Determine how many features fall within a defined distance from a point. This is common for service coverage, emergency response, or access planning.

```
# Find features within distance
con.sql("""
SELECT COUNT(*) as nearby_stations
FROM nyc_subway_stations
WHERE ST_DWithin(geom, ST_GeomFromText('POINT(583571 4506714)'), 500);
""")
```

Key Notes on ST_DWithin :

- Returns `TRUE` if two geometries are within a specified distance
- Units depend on the CRS (usually meters)
- More efficient than `ST_Distance()` when exact distance is unnecessary

25.10. Spatial Joins

Spatial joins are one of the most powerful operations in geospatial analysis. Unlike regular database joins that connect tables based on matching values (like IDs), spatial joins connect records based on their geographic relationships. This allows you to answer complex questions like “Which neighborhoods have subway stations?” or “What is the population of areas within 1km of hospitals?”

25.10.1. Understanding Spatial vs. Regular Joins

Regular Join: Connects records where column values match exactly

- Example: `customers.city_id = cities.id`

Spatial Join: Connects records where spatial relationships exist

- Example: `ST_Intersects(neighborhoods.geometry, subway_stations.geometry)`

25.10.2. Exploring Our Data

Before joining, it’s helpful to inspect the datasets:

```
con.sql("FROM nyc_neighborhoods SELECT * LIMIT 5;")
```

```
con.sql("FROM nyc_subway_stations SELECT * LIMIT 5;")
```

What we’re seeing in these tables: each table has a geometry column containing spatial data, additional attribute columns provide descriptive information, and the spatial join will combine these based on where features intersect geographically.

25.10.3. Your First Spatial Join

Let’s associate a subway station with the neighborhood it falls in:

```
# Find subway stations and their neighborhoods
con.sql("""
```

```
SELECT
    subways.name AS subway_name,
    neighborhoods.name AS neighborhood_name,
    neighborhoods.borname AS borough
FROM nyc_neighborhoods AS neighborhoods
JOIN nyc_subway_stations AS subways
ON ST_Intersects(neighborhoods.geom, subways.geom)
WHERE subways.NAME = 'Broad St';
""")
```

What's Happening Here:

1. We're joining two spatial tables (`neighborhoods` and `subways`)
2. `ST_Intersects` checks where station points fall within neighborhood polygons
3. We pull meaningful attributes from both tables (station name, neighborhood, borough)

25.10.4. Advanced Spatial Analysis

Beyond basic spatial joins, geospatial databases enable sophisticated analyses that combine spatial and non-spatial attributes to derive meaningful insights. In this section, we explore how spatial queries can help answer policy-relevant questions about transit access, demographic disparities, and infrastructure equity.

Before performing complex spatial operations, it's often useful to filter spatial features based on descriptive attributes. For example, we can isolate subway stations by their line color.

```
con.sql("""
SELECT DISTINCT COLOR FROM nyc_subway_stations;
""")
```

This helps identify available filters for downstream analysis (e.g., focusing on stations served by the RED line).

Let's identify which **RED line** subway stations intersect with NYC neighborhoods:

```
# Find RED subway stations and their neighborhoods
con.sql("""
SELECT
    subways.name AS subway_name,
    neighborhoods.name AS neighborhood_name,
    neighborhoods.borname AS borough
FROM nyc_neighborhoods AS neighborhoods
JOIN nyc_subway_stations AS subways
ON ST_Intersects(neighborhoods.geom, subways.geom)
WHERE subways.color = 'RED';
""")
```

This query tells us where red-line stations are located and which neighborhoods they serve—valuable for understanding transit coverage or planning service changes.

25.10.5. Distance-Based Analysis

Before zooming into specific locations, it helps to get a citywide demographic baseline. Here we compute the overall racial composition of NYC using census block data:

```
# Get baseline demographics
con.sql("""
SELECT
  100.0 * Sum(popn_white) / Sum(popn_total) AS white_pct,
  100.0 * Sum(popn_black) / Sum(popn_total) AS black_pct,
  Sum(popn_total) AS popn_total
FROM nyc_census_blocks;
""")
```

This acts as a benchmark for comparison with smaller areas of interest, such as those near subway stations. Suppose we want to evaluate racial composition within walking distance of subway stations that serve the **A train**. First, we identify which routes include 'A':

```
con.sql("""
SELECT DISTINCT routes
FROM nyc_subway_stations AS subways
WHERE strpos(subways.routes, 'A') > 0;
""")
```

Now we can calculate demographics for census blocks located within 200 meters of these A-train stations:

```
# Demographics within 200 meters of A-train
con.sql("""
SELECT
  100.0 * Sum(popn_white) / Sum(popn_total) AS white_pct,
  100.0 * Sum(popn_black) / Sum(popn_total) AS black_pct,
  Sum(popn_total) AS popn_total
FROM nyc_census_blocks AS census
JOIN nyc_subway_stations AS subways
ON ST_DWithin(census.geom, subways.geom, 200)
WHERE strpos(subways.routes, 'A') > 0;
""")
```

This type of proximity analysis can be used to study equitable access to public transit or to explore neighborhood-level disparities in infrastructure benefits.

25.10.6. Advanced Multi-Table Joins

To broaden the scope, let's compare demographic indicators across **all subway lines**. We first define a reference table listing subway routes:

```
# Create subway lines reference table
con.sql("""
CREATE OR REPLACE TABLE subway_lines ( route char(1) );
INSERT INTO subway_lines (route) VALUES
  ('A'),('B'),('C'),('D'),('E'),('F'),('G'),
  ('J'),('L'),('M'),('N'),('Q'),('R'),('S'),
  ('Z'),('1'),('2'),('3'),('4'),('5'),('6'),
  ('7');
""")
```

Next, we join this table with subway stations and census blocks to calculate demographics near each route:

```
# Analyze demographics by subway line
con.sql("""
SELECT
  lines.route,
  100.0 * Sum(popn_white) / Sum(popn_total) AS white_pct,
  100.0 * Sum(popn_black) / Sum(popn_total) AS black_pct,
  Sum(popn_total) AS popn_total
FROM nyc_census_blocks AS census
JOIN nyc_subway_stations AS subways
ON ST_DWithin(census.geom, subways.geom, 200)
JOIN subway_lines AS lines
ON strpos(subways.routes, lines.route) > 0
GROUP BY lines.route
ORDER BY black_pct DESC;
""")
```

This enables comparative analysis across transit corridors—essential for transit equity assessments, neighborhood investment planning, and targeting of public services.

25.11. Large-Scale Data Analysis

25.11.1. Analyzing the National Wetlands Inventory

The [National Wetlands Inventory \(NWI\)](#) provides detailed geospatial data on the type, size, and distribution of wetlands throughout the United States. This dataset, maintained by the U.S. Fish and Wildlife Service, is a critical resource for environmental scientists, land managers, and policy makers.

The original data provided by the US Fish and Wildlife Service (FWS) is in Shapefile and GeoDatabase formats, which have been converted to GeoParquet format and hosted on [Source Cooperative](#) [^source-cooperative]. The total size of the dataset is 75.8 GB.

Due to its large size, traditional desktop GIS tools struggle with analyzing the entire dataset efficiently. This is where **DuckDB** shines. With its support for **spatial extensions** and **remote file access**, DuckDB allows you to query large, cloud-hosted geospatial datasets with minimal setup.

```
# Connect and prepare for large-scale analysis
con = duckdb.connect()
con.install_extension("spatial")
con.load_extension("spatial")
```

Let's begin by querying wetlands data for a single state (e.g., Washington, D.C.).

```
# Analyze single state data
state = "DC" # Change to the US State of your choice
url = f"https://data.source.coop/giswqs/nwi/wetlands/{state}_Wetlands.parquet"
con.sql(f"SELECT * FROM '{url}'")
```

You can inspect the schema of the dataset to understand its structure:

```
# Inspect the table schema
con.sql(f"DESCRIBE FROM '{url}'")
```

25.11.2. Scaling Up: Nationwide Wetland Analysis

In this section, we will expand our analysis to include all 51 available state-level files in the [National Wetlands Inventory \(NWI\)](#) dataset, hosted by the [Source Cooperative](#). Each file represents wetlands data for a single United States state or territory.

The data is stored in a public S3 bucket using the efficient GeoParquet format. Thanks to DuckDB's ability to query files directly from cloud storage, we can perform national-scale analysis without needing to download the files locally.

25.11.2.1. Count the Total Number of Wetlands

```
con.sql(f"""
SELECT COUNT(*) AS Count
FROM 's3://us-west-2.opendata.source.coop/giswqs/nwi/wetlands/*.parquet'
""")
```

Result: There are over **38 million** wetland features in the United States, and the query completes in just a few seconds.

25.11.2.2. Count Wetlands by State

Since the NWI dataset does not include a state column, we can infer it from the filenames:

```
# Count wetlands by state using filename
df = con.sql(f"""
SELECT filename, COUNT(*) AS Count
FROM read_parquet('s3://us-west-2.opendata.source.coop/giswqs/nwi/wetlands/*.parquet', filename=true)
```

```
GROUP BY filename
ORDER BY COUNT(*) DESC;
""").df()
df.head()
```

We can now extract the state code from each filename:

```
# Extract state codes from filenames
count_df = con.sql(f"""
SELECT SUBSTRING(filename, LENGTH(filename) - 18, 2) AS State, COUNT(*) AS Count
FROM read_parquet('s3://us-west-2.opendata.source.coop/giswqs/nwi/wetlands/*.
parquet', filename=true)
GROUP BY State
ORDER BY COUNT(*) DESC;
""").df()
count_df.head(10)
```

To simplify future queries, let's save the result as a DuckDB table:

```
# Create a wetlands table from the DataFrame
con.sql("CREATE OR REPLACE TABLE wetlands AS FROM count_df")
con.sql("FROM wetlands")
```

25.11.3. Mapping Wetland Counts by State

To visualize these wetland counts, we join the `wetlands` table with a dataset of U.S. state boundaries, which includes geometries.

```
# Create states table with geometries
url = 'https://opengeos.org/data/us/us_states.parquet'
con.sql(
    f"""
CREATE OR REPLACE TABLE states AS
SELECT * FROM '{url}'
""")
)
```

We then join the wetlands count with the state geometries:

```
con.sql(
    """
SELECT * FROM states INNER JOIN wetlands ON states.id = wetlands.State
""")
)
```

```
# Join wetlands count with state geometries for visualization
file_path = "states_with_wetlands.geojson"
con.sql(f"""
    COPY (
    SELECT s.name, s.id, w.Count, s.geometry
    FROM states s
    JOIN wetlands w ON s.id = w.State
    ORDER BY w.Count DESC
    ) TO '{file_path}' WITH (FORMAT GDAL, DRIVER 'GeoJSON')
""")
```

Let's visualize the wetlands count on the map (Figure 123).

```
m = leafmap.Map()
m.add_data(
    file_path, column="Count", scheme="Quantiles", cmap="Greens",
    legend_title="Wetland Count"
)
m
```

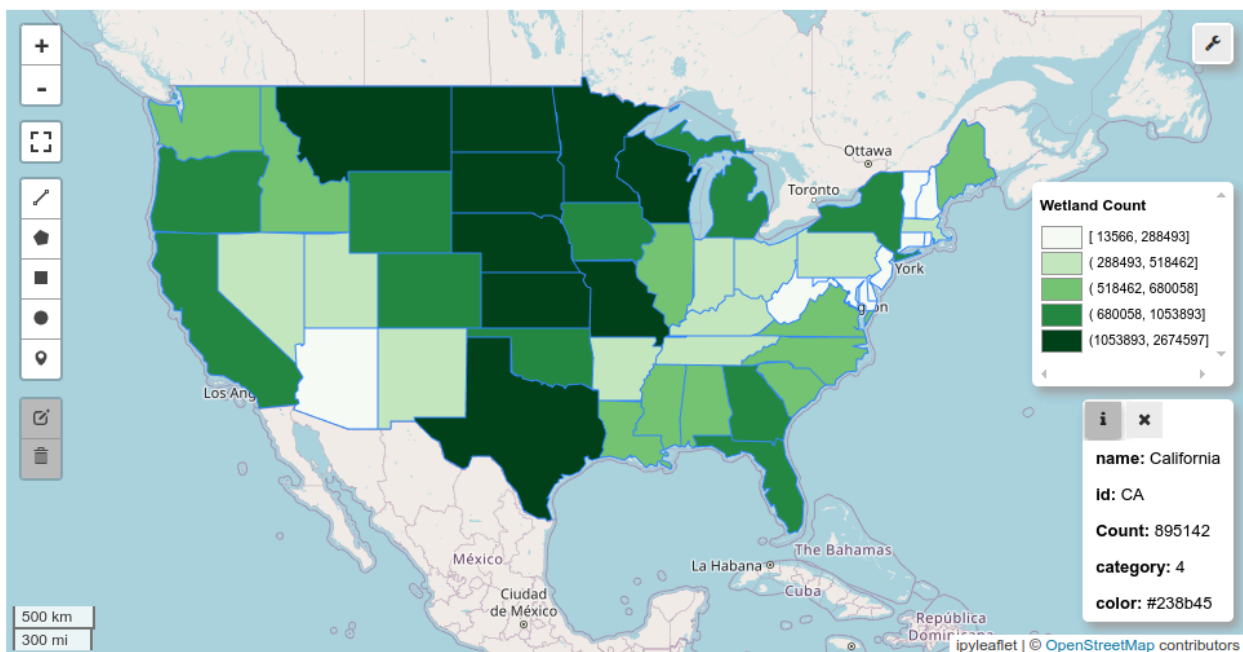


Figure 123: Choropleth map showing the number of wetlands per state.

25.11.4. Wetland Distribution Charts

25.11.4.1. Pie Chart of Wetlands by State

The pie chart below shows the relative proportion of wetlands in each U.S. state. This visualization helps identify which states contribute the most to the national wetland count ([Figure 124](#)).

```
leafmap.pie_chart(  
    count_df, "State", "Count", height=800, title="Number of Wetlands by State"  
)
```

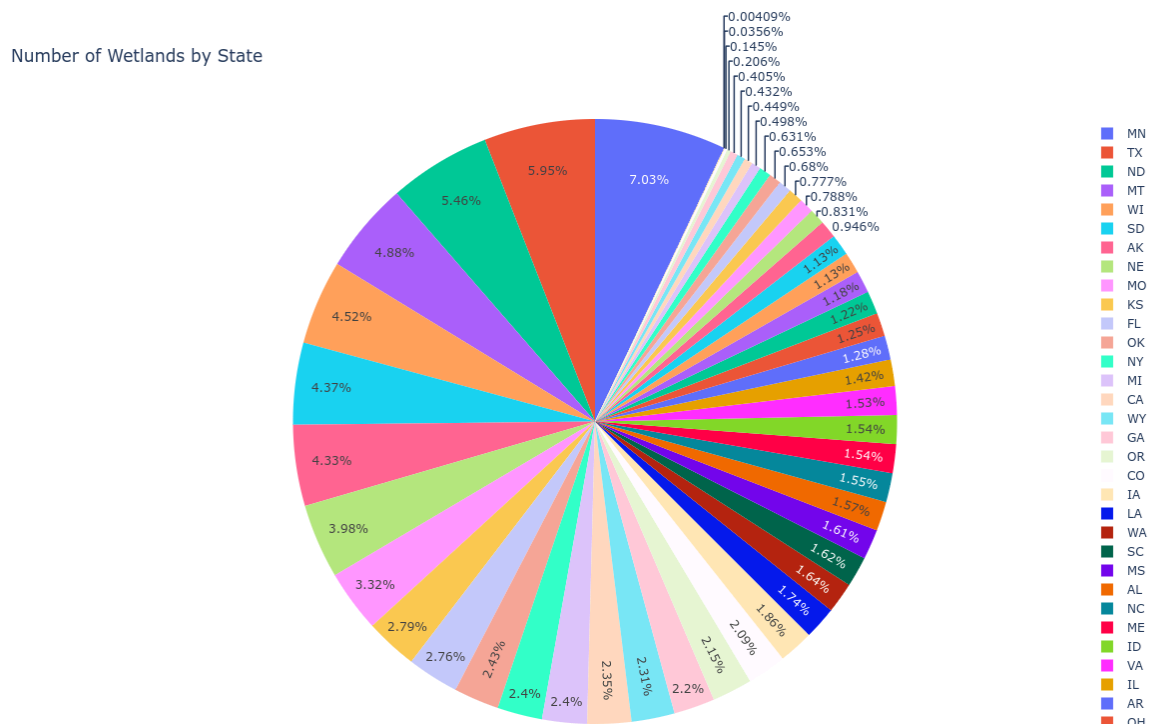


Figure 124: The pie chart shows the number of wetlands in each state.

25.11.4.2. Bar Chart of Wetlands by State

The bar chart provides an exact count of wetlands by state, allowing easier comparisons across states. It highlights outliers such as states with especially high or low wetland counts ([Figure 125](#)).

```
leafmap.bar_chart(count_df, "State", "Count", title="Number of Wetlands by  
State")
```

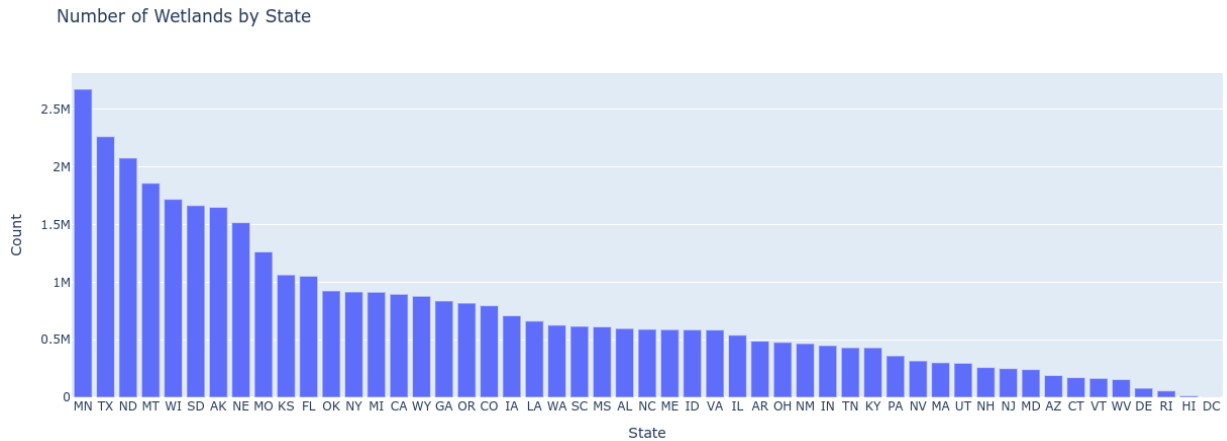


Figure 125: The bar chart shows the number of wetlands in each state.

25.11.5. Wetland Area Analysis

In addition to wetland counts, the total area covered by wetlands in each state gives a more comprehensive view of wetland distribution. Some states may have fewer wetlands that cover much larger areas.

25.11.5.1. Total Wetland Area in the U.S.

Let's calculate the total area of wetlands in the United States.

```
con.sql(
    f"""
    SELECT SUM(Shape_Area) / 1000000 AS Area_SqKm
    FROM 's3://us-west-2.opendata.source.coop/giswqs/nwi/wetlands/*.parquet'
    """
)
```

Result: Total wetland area is approximately **1,442,876 km²**

25.11.5.2. Wetland Area by State

```
area_df = con.sql(
    f"""
    SELECT SUBSTRING(filename, LENGTH(filename) - 18, 2) AS State, SUM(Shape_Area) /
    1000000 AS Area_SqKm
    FROM read_parquet('s3://us-west-2.opendata.source.coop/giswqs/nwi/wetlands/*.
    parquet', filename=true)
    GROUP BY State
    ORDER BY COUNT(*) DESC;
    """
).df()
area_df.head(10)
```

25.11.5.3. Pie Chart of Wetland Area by State

This chart represents the proportion of total U.S. wetland area located in each state. It emphasizes states that contribute most to the country's total wetland coverage ([Figure 126](#)).

```
leafmap.pie_chart(  
  area_df, "State", "Area_SqKm", height=850, title="Wetland Area by State"  
)
```

Wetland Area by State

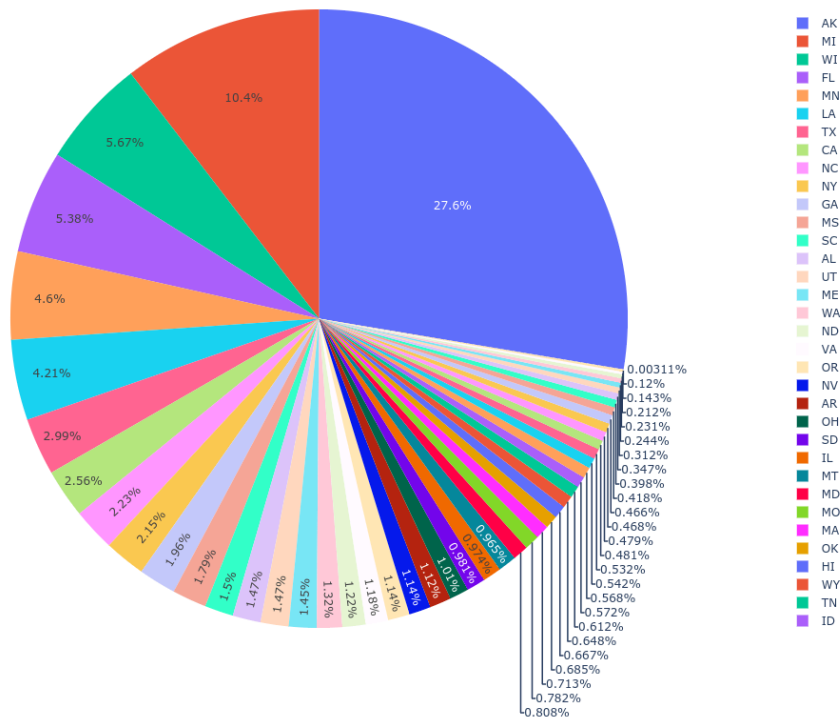


Figure 126: The pie chart shows the area of wetlands in each state.

25.11.5.4. Bar Chart of Wetland Area by State

This chart visualizes the total wetland area per state in square kilometers, allowing for precise comparisons of wetland extent ([Figure 127](#)).

```
leafmap.bar_chart(area_df, "State", "Area_SqKm", title="Wetland Area by State")
```

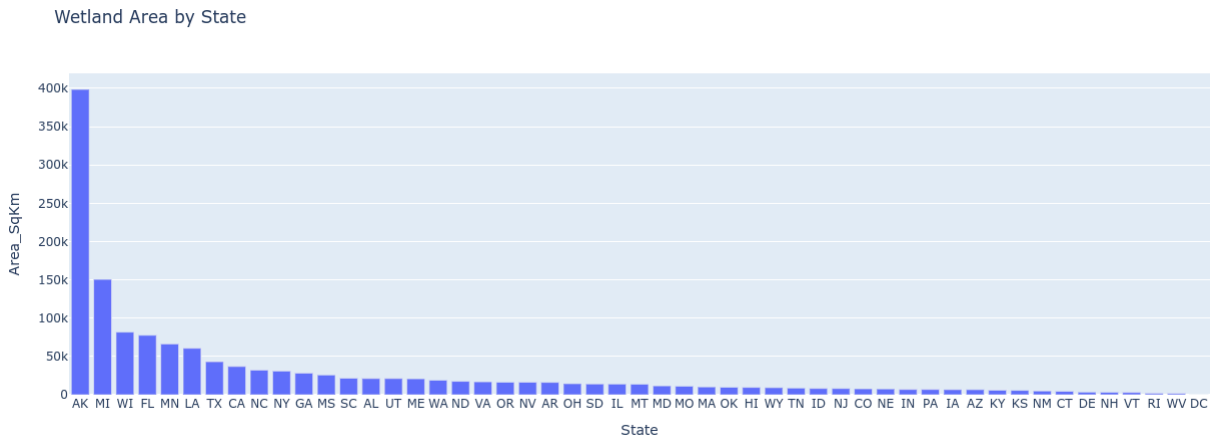


Figure 127: The bar chart shows the area of wetlands in each state.

25.12. Key Takeaways

This chapter introduced DuckDB as a game-changing tool for geospatial analytics. By combining the power of SQL with modern analytical database architecture, DuckDB makes high-performance spatial analysis accessible to anyone familiar with basic database concepts.

25.12.1. Fundamental Concepts You’ve Learned

What DuckDB Is and Why It Matters: DuckDB is an embedded analytical database that excels at the types of operations common in geospatial work—aggregations, joins, and analysis across large datasets. Unlike traditional databases, it requires no server setup and can work directly with your data files.

Zero-ETL Data Access: One of DuckDB’s most revolutionary features is its ability to query data directly from files (CSV, JSON, Parquet, GeoJSON, Shapefiles) without importing them first. This eliminates the traditional Extract-Transform-Load bottleneck that often slows down spatial analysis projects.

SQL for Spatial Analysis: SQL provides a powerful, declarative way to ask questions about spatial data. Instead of writing complex loops and conditionals, you can express spatial queries in natural language-like statements that are both readable and efficient.

Geometry Fundamentals: Understanding the basic geometry types (Points, LineStrings, Polygons) and how they represent real-world features is essential for all spatial analysis. DuckDB’s support for these geometries follows OGC standards, ensuring compatibility with other spatial tools.

Spatial Relationships: The ability to test how geometries relate to each other (intersects, contains, within, distance-based) forms the foundation of spatial analysis. These relationships let you answer fundamental geographic questions.

Spatial Joins: Perhaps the most powerful concept in spatial analysis, spatial joins allow you to combine data from different sources based on their geographic relationships rather than matching IDs or keys.

25.12.2. Why This Approach is Powerful

Performance: DuckDB’s columnar storage and vectorized execution make it exceptionally fast for analytical queries, even on large spatial datasets.

Simplicity: The embedded architecture means you can start analyzing spatial data immediately without complex database setup or administration.

Integration: Seamless integration with Python’s ecosystem (pandas, GeoPandas, visualization libraries) allows you to combine SQL’s power with Python’s flexibility.

Accessibility: By using familiar SQL syntax with spatial extensions, DuckDB makes advanced spatial analysis techniques accessible to anyone with basic database knowledge.

25.12.3. Building Your Spatial Analysis Skills

The concepts introduced in this chapter—data import, basic SQL operations, geometry handling, spatial relationships, and visualization integration—form the foundation for more advanced spatial analysis techniques. As you continue your geospatial journey, these fundamentals will serve as building blocks for increasingly sophisticated analyses.

25.13. Exercises

Sample Datasets:

The following datasets are used in the exercises. You don’t need to download them manually, they can be accessed directly from the notebook.

- [nyc_subway_stations.tsv](https://opengeos.org/data/duckdb/nyc_subway_stations.tsv)¹³¹
- [nyc_neighborhoods.tsv](https://opengeos.org/data/duckdb/nyc_neighborhoods.tsv)¹³²

25.13.1. Exercise 1: Creating Tables

Create a database, then write a SQL query to create a table named `nyc_subway_stations` and load the data from the file `nyc_subway_stations.tsv` into it. Similarly, create a table named `nyc_neighborhoods` and load the data from the file `nyc_neighborhoods.tsv` into it.

25.13.2. Exercise 2: Column Filtering

Write a SQL query to display the `ID`, `NAME`, and `BOROUGH` of each subway station in the `nyc_subway_stations` dataset.

25.13.3. Exercise 3: Row Filtering

Write a SQL query to find all subway stations in the `nyc_subway_stations` dataset that are located in the borough of Manhattan.

¹³¹https://opengeos.org/data/duckdb/nyc_subway_stations.tsv

¹³²https://opengeos.org/data/duckdb/nyc_neighborhoods.tsv

25.13.4. Exercise 4: Sorting Results

Write a SQL query to list the subway stations in the `nyc_subway_stations` dataset in alphabetical order by their names.

25.13.5. Exercise 5: Unique Values

Write a SQL query to find the distinct boroughs represented in the `nyc_subway_stations` dataset.

25.13.6. Exercise 6: Counting Rows

Write a SQL query to count the number of subway stations in each borough in the `nyc_subway_stations` dataset.

25.13.7. Exercise 7: Aggregating Data

Write a SQL query to list the number of subway stations in each borough, sorted in descending order by the count.

25.13.8. Exercise 8: Joining Tables

Write a SQL query to join the `nyc_subway_stations` and `nyc_neighborhoods` datasets on the borough name, displaying the subway station name and the neighborhood name.

25.13.9. Exercise 9: String Manipulation

Write a SQL query to display the names of subway stations in the `nyc_subway_stations` dataset that contain the word “St” in their names.

25.13.10. Exercise 10: Filtering with Multiple Conditions

Write a SQL query to find all subway stations in the `nyc_subway_stations` dataset that are in the borough of Brooklyn and have routes that include the letter “R”.

25.13.11. Exercise 11: Basic Setup and Data Loading

Connect to a DuckDB database and install the `httpfs` and `spatial` extensions

25.13.12. Exercise 12: Spatial Relationships Analysis

Download the [Admin 0 – Countries](#)¹³³ vector dataset from Natural Earth using the `leafmap.download_file()` function.

25.13.13. Exercise 13: Advanced Spatial Joins

Create a new table in your database called `countries` and load the data from the downloaded country shapefile into it.

Calculate the total population of all countries in the database using the `POP_EST` column.

Show the top 10 countries with the largest population.

Select countries in Europe with a population greater than 10 million and order them by population in descending order.

Save the results of the previous query as a new table called `europe`.

Export the `europe` table as a GeoJSON file.

25.13.14. Exercise 14: Data Import and Export

Create a table called `text_zones` and load the data from the [taxi_zones.parquet](#)¹³⁴ into it.

¹³³<https://www.naturalearthdata.com/downloads/10m-cultural-vectors>

¹³⁴https://beta.source.coop/cholmes/nyc-taxi-zones/taxi_zones.parquet

Find out the unique values in the `borough` column and order them alphabetically.

Export the `text_zones` table as a parquet file.

25.13.15. Exercise 15: Large-Scale Analysis

Explore the [Google Open Buildings](#)¹³⁵ and select a country of your choice with a relatively small number of buildings (i.e., small file size). Get the three-character country code and replace `[COUNTRY_NAME]` in the following path with the country code. Use it to load all the parquet files for the selected country into a new table called `buildings`.

```
s3://us-west-2.opendata.source.coop/google-research-open-buildings/v2/geoparquet-admin1/country=[COUNTRY_NAME]/*.parquet
```

Find out the number of buildings in the selected country.

Find out the total area of all buildings in the selected country.

Export the `buildings` table as a GeoPackage file.

¹³⁵<https://beta.source.coop/cholmes/google-open-buildings/v2/geoparquet-admin1>

Chapter 26. Geospatial Data Processing with GDAL and OGR

26.1. Introduction

If you’ve ever opened a satellite image in QGIS, converted a shapefile to GeoJSON, or transformed coordinates between different map projections, you’ve almost certainly used GDAL—even if you didn’t realize it. [GDAL](https://gdal.org)¹³⁶ (Geospatial Data Abstraction Library) and its vector counterpart OGR form the invisible foundation beneath virtually every piece of geospatial software you encounter.

Think of GDAL as the universal translator for geospatial data. Just as Google Translate can convert between dozens of human languages, GDAL can read and write over 200 different raster formats and 80+ vector formats. Whether you’re working with NASA satellite imagery stored in HDF5 format, vintage survey data in an obscure proprietary format, or modern web-friendly GeoJSON files, GDAL provides a consistent, unified way to access and manipulate all of this data.

This universality makes GDAL incredibly powerful, but it can also make it seem intimidating at first. The library handles an enormous range of data types and formats, which means it has many functions and options. However, understanding the core concepts and most common operations will give you the foundation to handle the vast majority of geospatial data challenges you’ll encounter.

GDAL operates on a simple but powerful principle: it provides drivers for different data formats and presents a unified interface for working with spatial data regardless of the underlying format. This means you can use the same basic approach whether you’re reading a GeoTIFF image or a NetCDF climate dataset, or whether you’re working with an ESRI Shapefile or a PostGIS database.

The relationship between GDAL and higher-level Python libraries is crucial to understand. Libraries like GeoPandas and Rasterio that we’ve explored in previous chapters are actually built on top of GDAL. They provide more Pythonic, user-friendly interfaces while GDAL handles the heavy lifting of format support and data transformation behind the scenes. This chapter teaches you to work directly with GDAL, giving you deeper control and understanding of what happens beneath those convenient abstractions.

26.1.1. When to Use GDAL Directly

While the higher-level libraries are usually more convenient for day-to-day analysis, there are times when working directly with GDAL is necessary or advantageous. You’ll want to use GDAL directly when you need maximum control over data reading and writing operations, when working with formats that aren’t well-supported by other libraries, when performing complex data transformations that require fine-tuning for optimal performance, or when building automated processing pipelines that need to handle diverse data formats reliably.

GDAL also excels at batch processing operations where you need to apply the same transformation to many files efficiently. Its command-line tools are particularly powerful for this kind of work, allowing you to process entire directories of data with simple shell scripts or automated workflows.

26.2. Learning Objectives

By the end of this chapter, you should be able to:

- Understand GDAL’s architecture and the relationship between GDAL (raster) and OGR (vector)

¹³⁶<https://gdal.org>

- Use GDAL to read and inspect raster and vector datasets
- Extract metadata and spatial reference information from geospatial files
- Perform coordinate transformations and reprojections
- Convert between different geospatial data formats
- Perform raster and vector analysis
- Manage fields and layers
- Tiling and merging raster and vector data
- Perform terrain analysis

26.3. Installation and Setup

Before we begin working with GDAL, we need to install it along with the `pygis` package, which provides additional Python geospatial tools that work well with GDAL. The conda-forge channel provides the most reliable GDAL installation with all the necessary drivers and dependencies.

```
conda install -c conda-forge gdal pygis
```

26.4. Sample Datasets

For this chapter, we'll work with a comprehensive set of sample datasets that demonstrate different aspects of GDAL functionality. These datasets include both raster and vector data in various formats, representing common real-world scenarios you'll encounter in geospatial analysis.

The sample data includes building footprints and masks, digital elevation models, satellite imagery, and administrative boundaries. Each dataset serves a specific purpose in demonstrating different GDAL capabilities, from simple format conversion to complex geometric operations. These sample datasets are available in the [opengeos/datasets](https://github.com/opengeos/datasets) repository¹³⁷.

First, let's download the sample datasets:

```
wget https://github.com/opengeos/datasets/releases/download/gdal/gdal_sample_data.zip
```

Unzip the downloaded file:

```
unzip gdal_sample_data.zip
```

26.5. Understanding Your Data

Before performing any operations on geospatial data, you need to understand what you're working with. GDAL provides powerful tools for inspecting both raster and vector datasets, revealing crucial information about coordinate systems, data types, spatial extents, and internal structure.

¹³⁷<https://github.com/opengeos/datasets/releases/tag/gdal>

26.5.1. Examining Raster Data

The `gdalinfo` command is your primary tool for understanding raster datasets. It reveals everything from basic dimensions and data types to complex metadata and coordinate reference systems. Understanding this information is essential before performing any transformations or analysis.

```
gdalinfo dem.tif
```

This command reveals the raster's dimensions, coordinate reference system, geotransform parameters (which define how pixel coordinates map to real-world coordinates), and any embedded metadata. Pay particular attention to the coordinate system information, as this determines how the data relates to locations on Earth.

26.5.2. Examining Vector Data

For vector data, OGR provides the `ogrinfo` command, which reveals layer structure, attribute schemas, coordinate systems, and feature counts. Vector data can be more complex than raster data because it often contains multiple layers and rich attribute information.

```
ogrinfo buildings.geojson
```

The basic `ogrinfo` command shows you the available layers and their basic properties. However, you often need more detailed information about the layer contents and structure.

```
ogrinfo buildings.geojson -al -so
```

The `-al` flag shows information about all layers, while `-so` provides a summary only, which is faster for large datasets. This combination gives you the essential information about field names, data types, coordinate systems, and spatial extents without loading all the feature data.

26.6. Coordinate Transformation

One of GDAL's most powerful capabilities is coordinate transformation—the ability to convert data between different coordinate reference systems. This is fundamental to geospatial work because data from different sources often uses different projections or coordinate systems.

26.6.1. Reprojecting Raster Data

Raster reprojection involves transforming both the coordinate system and resampling the pixel values to fit the new coordinate grid. The `gdalwarp` command handles this complex process automatically, choosing appropriate resampling methods and handling edge cases.

```
gdalwarp -t_srs EPSG:4326 dem.tif dem_4326.tif
```

This command reprojects the digital elevation model from whatever coordinate system it's currently in to WGS84 geographic coordinates (EPSG:4326). The transformation process involves several steps: GDAL

determines the source coordinate system, calculates the transformation parameters, creates a new pixel grid in the target coordinate system, and resamples the elevation values to populate the new grid.

26.6.2. Reprojecting Vector Data

Vector reprojection is generally more straightforward than raster reprojection because it involves transforming coordinate pairs rather than resampling continuous data. However, it's still important to understand what's happening during the transformation.

```
ogr2ogr -t_srs EPSG:3857 buildings_3857.gpkg buildings.geojson
```

This command transforms building footprints from their original coordinate system to Web Mercator (EPSG:3857), which is commonly used for web mapping applications. The transformation changes the coordinate values but preserves the geometric relationships between features.

26.7. Format Conversion

GDAL's format conversion capabilities are among its most frequently used features. The ability to convert between different geospatial formats ensures interoperability between different software systems and workflows.

26.7.1. Converting Raster Formats

Raster format conversion involves more than just changing file extensions. Different formats have different capabilities, compression options, and metadata storage systems. Understanding these differences helps you choose the right format for your specific needs.

```
gdal_translate dem.tif dem.img
```

The `gdal_translate` command converts the GeoTIFF elevation model to ERDAS IMAGINE format. While both formats store similar information, they have different internal structures and capabilities. GDAL handles these differences transparently while preserving as much information as possible.

26.7.2. Converting Vector Formats

Vector format conversion is similarly complex, as different formats support different feature types, attribute storage systems, and spatial capabilities. Some formats support multiple layers while others don't, and some have limitations on field names or data types.

```
ogr2ogr buildings.gpkg buildings.geojson
```

This conversion transforms GeoJSON data to [GeoPackage](https://www.geopackage.org)¹³⁸ format. GeoPackage is often preferable for analysis because it supports spatial indexing, can store multiple layers in a single file, and handles attribute data more efficiently than GeoJSON.

¹³⁸<https://www.geopackage.org>

```
ogr2ogr las_vegas_bbox.fgb las_vegas_bbox.geojson
```

[FlatGeobuf](#) (FGB)¹³⁹ is an increasingly popular format for large vector datasets because it provides very fast access to spatial data through efficient internal indexing and a columnar storage structure.

26.8. Clipping and Masking

Spatial clipping operations allow you to extract subsets of data based on geographic boundaries. This is essential for focusing analysis on specific areas of interest and reducing data volumes for processing efficiency.

26.8.1. Clipping Raster with Vector Boundaries

Raster clipping involves using vector boundaries to mask or crop raster data. This operation is fundamental for extracting data for specific study areas or administrative boundaries.

```
gdalwarp -cutline las_vegas_bbox.geojson -crop_to_cutline landsat.tif  
las_vegas.tif
```

The `-cutline` parameter specifies the vector boundary to use for clipping, while `-crop_to_cutline` ensures that the output raster is cropped to the minimum bounding rectangle of the clipping boundary. This creates a smaller, more focused dataset that contains only the pixels within the specified boundary.

26.8.2. Clipping Vector Data

Vector clipping can use either bounding boxes or other vector geometries as clipping boundaries. The choice depends on whether you need precise geometric clipping or just a rectangular subset.

```
ogr2ogr -clipsrc las_vegas_bbox.geojson las_vegas_roads_clipped.geojson  
las_vegas_roads.geojson
```

This command clips road features to the Las Vegas boundary, retaining only those road segments that intersect with the boundary polygon. Features that cross the boundary are split at the boundary line.

Let's examine the bounding box coordinates so we can also demonstrate coordinate-based clipping:

```
ogrinfo las_vegas_bbox_4326.geojson -al -so
```

Now we can perform the same clipping operation using explicit coordinate bounds:

```
ogr2ogr -clipsrc -115.387634 35.943333 -114.883495 36.359161  
las_vegas_roads_clipped.geojson las_vegas_roads.geojson
```

This approach is useful when you know the exact coordinates of your study area or when working with regular grids or tiles.

¹³⁹<https://flatgeobuf.org>

26.9. Raster Analysis and Calculations

GDAL provides powerful capabilities for raster analysis, including band extraction, mathematical operations, and index calculations. These operations form the foundation for more complex remote sensing and spatial analysis workflows.

26.9.1. Working with Individual Bands

Multi-band raster datasets, such as satellite imagery, often require analysis of individual spectral bands. GDAL makes it easy to extract specific bands for focused analysis.

```
gdal_translate -b 5 landsat.tif nir.tif
gdal_translate -b 4 landsat.tif red.tif
gdal_translate -b 3 landsat.tif green.tif
```

These commands extract the near-infrared, red, and green bands from a Landsat image. Each band captures different wavelengths of light, providing information about different surface properties. The near-infrared band is particularly useful for vegetation analysis because healthy vegetation strongly reflects near-infrared light.

26.9.2. Performing Band Mathematics

Mathematical operations between raster bands enable the calculation of vegetation indices, ratios, and other derived products that reveal information not visible in individual bands.

```
gdal_calc.py \
-A nir.tif \
-B red.tif \
--outfile=ndvi.tif \
--calc="(A.astype(float) - B) / (A + B + 1e-6)" \
--type=Float32 \
--NoDataValue=-9999
```

This calculation computes the Normalized Difference Vegetation Index (NDVI), a widely used indicator of vegetation health and density. The formula $(\text{NIR} - \text{Red}) / (\text{NIR} + \text{Red})$ produces values between -1 and 1 , where higher values indicate healthier, denser vegetation. The small epsilon value ($1e-6$) prevents division by zero in areas where both bands have zero values.

Sometimes calculated indices need to be constrained to their theoretical ranges:

```
gdal_calc.py \
-A ndvi.tif \
--outfile=ndvi_clipped.tif \
--calc="(A < -1)*-1 + (A > 1)*1 + ((A >= -1) * (A <= 1))*A" \
--type=Float32 \
--NoDataValue=-9999 \
--overwrite
```

This expression clamps NDVI values to the valid range of -1 to 1 , which can be necessary when working with noisy data or when calculation errors produce out-of-range values.

You can also create binary masks based on threshold values:

```
gdal_calc.py -A ndvi_clipped.tif --outfile=vegetation.tif --calc="A>0.3"
```

This creates a binary mask where pixels with NDVI greater than 0.3 are marked as vegetation. Such masks are useful for area calculations and as inputs to other analysis processes.

Managing NoData values is crucial in raster analysis:

```
gdal_translate -a_nodata 0 vegetation.tif vegetation_bin.tif
```

```
gdal_edit.py -a_nodata 0 vegetation.tif
```

These commands set the NoData value for the binary vegetation mask, ensuring that zero values are properly treated as “no vegetation” rather than missing data.

26.10. Converting Between Raster and Vector

The ability to convert between raster and vector representations of spatial data is fundamental to many geospatial workflows. Each representation has advantages: rasters are efficient for continuous phenomena and spatial analysis, while vectors are better for discrete objects and attribute management.

26.10.1. Vectorization

Vectorization converts raster data into vector polygons, typically grouping adjacent pixels with the same value into discrete geometric features.

```
gdal_polygonize.py building_masks.tif building_masks.gpkg
```

This process traces the boundaries between different pixel values in the building mask raster, creating polygon features that represent individual buildings or groups of connected building pixels. The resulting vector data can then be analyzed using traditional GIS operations like area calculations, spatial joins, and topology analysis.

26.10.2. Rasterization

Rasterization converts vector data into raster format, which can be useful for spatial analysis, modeling, or integration with other raster datasets.

```
gdal_rasterize -a uid -tr 0.6 0.6 -l buildings buildings_3857.gpkg buildings.tif
```

This command rasterizes building polygons, using the ‘uid’ attribute to assign pixel values. The `-tr` parameter sets the target resolution (0.6×0.6 meters), which should be chosen based on the scale of your analysis and the precision of your source data.

Alternatively, you can create a simple presence/absence mask:

```
gdal_rasterize -burn 1 -tr 0.6 0.6 -l buildings buildings_3857.gpkg  
buildings2.tif
```

The `-burn` option assigns a fixed value (1) to all pixels that intersect building polygons, creating a binary mask that shows where buildings are present.

26.11. Geometry Processing

GDAL provides sophisticated tools for processing vector geometries, including simplification, dissolution, and splitting operations that are essential for data cleaning and analysis preparation.

26.11.1. Simplifying Complex Geometries

Geometric simplification reduces the number of vertices in polygons and polylines while preserving their essential shape characteristics. This is important for reducing file sizes, improving display performance, and meeting data sharing requirements.

```
ogr2ogr -f GPKG -t_srs EPSG:26911 -simplify 1 simplified.gpkg building_masks.gpkg
```

This command combines reprojection with simplification, removing vertices that are less than 1 meter apart. The simplification tolerance should be chosen based on your data accuracy requirements and intended use. Larger tolerance values create simpler geometries but may lose important shape details.

26.11.2. Dissolving Features by Attributes

Dissolution combines adjacent features that share common attribute values, creating larger, unified features. This operation is essential for creating summary datasets and reducing geometric complexity.

First, let's examine the structure of our dataset:

```
ogrinfo -al -so us_states.geojson
```

Now we can dissolve the state boundaries by country, combining all states that belong to the same country:

```
ogr2ogr -dialect sqlite -sql "SELECT ST_Union(geometry), country FROM us_states  
GROUP BY country" us_states_dissolved.gpkg us_states.geojson
```

This SQL query uses PostGIS-style spatial functions to union geometries that share the same country attribute. The result is a simplified dataset with one feature per country rather than one per state.

26.11.3. Exploding Multi-part Geometries

Sometimes you need to convert multi-part geometries (like MultiPolygons) into individual single-part features for analysis or visualization purposes.


```
ogr2ogr -explodecollections hawaii_parts.geojson hawaii.geojson
```

This operation converts each part of a multi-part geometry into a separate feature, which can be useful for individual island analysis or when working with software that doesn't handle multi-part geometries well.

26.12. Managing Fields and Layers

Effective attribute management is crucial for maintaining clean, efficient datasets. GDAL provides comprehensive tools for selecting, renaming, and modifying the structure of your vector data.

26.12.1. Selecting Specific Fields

When working with datasets that have many attributes, you often need only a subset of the fields for your analysis. Selecting specific fields reduces file sizes and simplifies data handling.

```
ogr2ogr -select id,name us_states_select.geojson us_states.geojson
```

This creates a new dataset containing only the 'id' and 'name' fields from the original state boundaries. This approach is particularly valuable when sharing data or when creating focused datasets for specific analyses.

26.12.2. Renaming Layers

Layer names should be descriptive and consistent with your project's naming conventions. The ability to rename layers during conversion helps maintain organized datasets.

```
ogr2ogr -nln states us_states_rename.gpkg us_states_select.geojson
```

The `-nln` (new layer name) parameter creates a layer called 'states' in the output GeoPackage, which is more descriptive than the default layer name derived from the filename.

26.12.3. Adding New Fields

Sometimes you need to add calculated fields or additional attributes to existing datasets. While this can be done during analysis, it's often useful to modify the dataset structure directly.

```
ogrinfo us_states_rename.gpkg -sql "ALTER TABLE states ADD COLUMN area DOUBLE"
```

This SQL command adds a new 'area' field to store calculated area values. The field can then be populated using additional SQL commands or analysis operations.

26.13. Tiling and Data Management

For large datasets, tiling and merging operations are essential for efficient processing and storage. These operations allow you to work with datasets that are too large to process as single files or to combine multiple datasets into unified products.

26.13.1. Creating Raster Tiles

Tiling large rasters improves processing efficiency and enables parallel processing of different parts of a dataset.

```
mkdir -p tiles
```

```
gdal_retile.py -targetDir tiles -ps 512 512 -co "TILED=YES" landsat.tif
```

This command splits the Landsat image into 512x512 pixel tiles, stored in the ‘tiles’ directory. The `-co "TILED=YES"` option creates internally tiled GeoTIFF files, which provide better performance for many operations.

26.13.2. Merging Raster Data

When you need to combine multiple raster files into a single dataset, GDAL provides several approaches depending on your specific requirements.

```
gdal_merge.py -o landsat_merged.tif -of GTiff tiles/*.tif
```

The `gdal_merge.py` script is straightforward for simple merging operations, combining all the tiles back into a single file. This approach works well for datasets that were previously tiled.

```
gdalwarp -of GTiff tiles/*.tif landsat_merged2.tif
```

Using `gdalwarp` for merging provides more control over the process and can handle overlapping areas more sophisticated ways, including blending and resampling options.

26.13.3. Working with Multiple Vector Files

Vector datasets often need to be split for distribution or analysis, then combined for final products.

Creating individual files for each state demonstrates the splitting process:

```
mkdir -p states
```

```
ogrinfo -ro -al -geom=NO us_states.geojson | grep "id (" | awk '{print $4}' |  
while read id; do  
    ogr2ogr -f GeoJSON "states/${id}.geojson" us_states.geojson -where "id =  
'${id}'"  
done
```

This script creates individual GeoJSON files for each state, naming each file with the state’s ID. This approach is useful for distributed processing or when different users need access to specific geographic areas.

Merging vector files back together requires careful attention to maintaining consistent schemas and coordinate systems:

```
ogr2ogr -f "ESRI Shapefile" us_states_merged.shp us_states.geojson
```

```
ogr2ogr -f "ESRI Shapefile" -update -append \  
us_states_merged.shp hawaii.geojson -nln us_states_merged
```

The `-update -append` flags add features from the Hawaii dataset to the existing merged shapefile, combining datasets from different sources into a unified product.

26.14. Advanced Raster Processing

GDAL provides sophisticated tools for raster data management, including resampling, band composition, and data quality improvement operations.

26.14.1. Resampling and Resolution Management

Changing raster resolution is a fundamental operation in spatial analysis, whether you're matching datasets to common resolutions or reducing data volumes for faster processing.

```
gdalwarp -tr 100 100 -r average dem.tif dem_100m.tif
```

This command resamples the digital elevation model to 100-meter resolution using average resampling. The choice of resampling method is important: 'average' is appropriate for continuous data like elevation, while 'nearest' is better for categorical data like land cover classifications.

26.14.2. Creating Band Composites

Multi-band raster composites are essential for remote sensing analysis and visualization. GDAL provides several approaches for combining individual bands into multi-band products.

```
gdal_merge.py -separate -o composite.tif nir.tif red.tif green.tif
```

The `-separate` flag ensures that each input file becomes a separate band in the output composite, maintaining the spectral information from each individual band.

For large datasets, Virtual Raster (VRT) files provide an efficient alternative:

```
gdalbuildvrt -separate stack.vrt nir.tif red.tif green.tif  
gdal_translate stack.vrt composite.tif -co COMPRESS=DEFLATE
```

VRT files don't store pixel data directly but instead reference the source files, making them very efficient for creating composites from large datasets. The final translation step creates the actual composite file with compression.

26.14.3. Handling Missing Data

Real-world raster datasets often contain gaps or missing values that need to be addressed before analysis.

```
gdal_fillnodata.py -md 5 -of GTiff dem.tif filled_dem.tif
```

This interpolation algorithm fills small gaps in the elevation data using surrounding values. The `-md 5` parameter limits gap-filling to areas smaller than 5 pixels, preventing the algorithm from filling large data voids where interpolation would be unreliable.

Properly defining NoData values is crucial for accurate analysis:

```
gdal_translate -a_nodata 0 dem.tif dem_nodata.tif
```

This ensures that zero values in the elevation data are properly recognized as missing data rather than sea level elevations.

26.14.4. Cloud Optimized GeoTIFF

Modern geospatial workflows increasingly rely on cloud-optimized formats that support efficient web-based access and streaming.

```
gdal_translate dem.tif dem_cog.tif -of COG -co COMPRESS=DEFLATE
```

Cloud Optimized GeoTIFF (COG) format includes internal tiling and overviews that enable efficient access to subsets of large raster datasets over network connections, making it ideal for web mapping and cloud-based analysis.

26.15. Terrain Analysis

Digital elevation models are fundamental to many geospatial analyses, and GDAL provides comprehensive tools for deriving terrain information from elevation data.

26.15.1. Computing Slope

Slope analysis reveals the steepness of terrain, which is crucial for hydrology, erosion modeling, and habitat analysis.

```
gdaldem slope dem.tif slope.tif \  
-of GTiff \  
-compute_edges
```

By default, GDAL computes slope in degrees, measuring the angle between the surface and horizontal. The `-compute_edges` flag ensures that slope values are calculated even at the edges of the dataset, preventing data loss around the boundaries.

For certain applications, slope as percent rise is more intuitive:

```
gdaldem slope dem.tif slope_percent.tif \  
-of GTiff \  
-compute_edges \  
-p
```

Percent slope represents the vertical rise over horizontal distance, expressed as a percentage. A 45-degree slope equals 100% slope.

26.15.2. Computing Aspect

Aspect analysis determines the direction that slopes face, which affects solar exposure, vegetation patterns, and erosion processes.

```
gdaldem aspect dem.tif aspect.tif \  
-of GTiff \  
-compute_edges
```

The output aspect is measured in degrees from 0° (North) clockwise to 360°, with flat areas assigned a value of -1. This information is essential for solar radiation modeling and ecological analysis.

26.15.3. Creating Hillshade Visualizations

Hillshading creates three-dimensional visualizations of terrain by simulating shadows cast by a hypothetical light source ([Figure 128](#)).

```
gdaldem hillshade dem.tif hillshade.tif
```



Figure 128: The hillshade visualization of the digital elevation model.

Standard hillshading assumes a single light source, typically positioned in the northwest. This creates dramatic relief visualization but can sometimes obscure details in areas facing away from the light source (Figure 129).

```
gdaldem hillshade dem.tif multidirectional_hillshade.tif -multidirectional
```

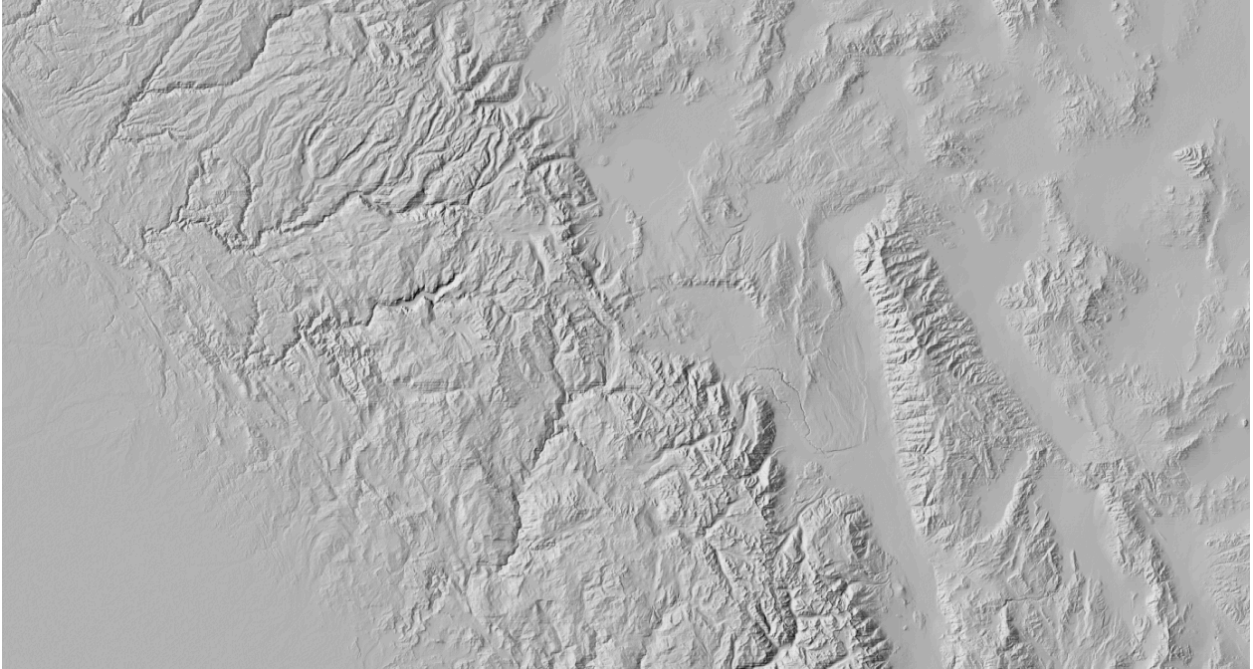


Figure 129: The multidirectional hillshade visualization of the digital elevation model.

Multidirectional hillshading uses multiple light sources to create more balanced illumination, revealing terrain details that might be lost in standard single-source hillshading.

26.15.4. Creating Color Relief Maps

Color relief maps use color gradients to represent elevation values, making terrain patterns more accessible to visual interpretation.

```
cat << EOF > colormap.txt
500,51,51,153
1000,3,147,249
1500,37,211,109
2000,181,240,138
2500,218,208,133
3000,146,115,94
4000,183,163,159
5000,255,255,255
EOF
```


This colormap defines color transitions from low elevations (blue) through middle elevations (green/yellow) to high elevations (brown/white), creating an intuitive representation of terrain (Figure 130).

```
gdaldem color-relief dem.tif colormap.txt color_relief.tif
```

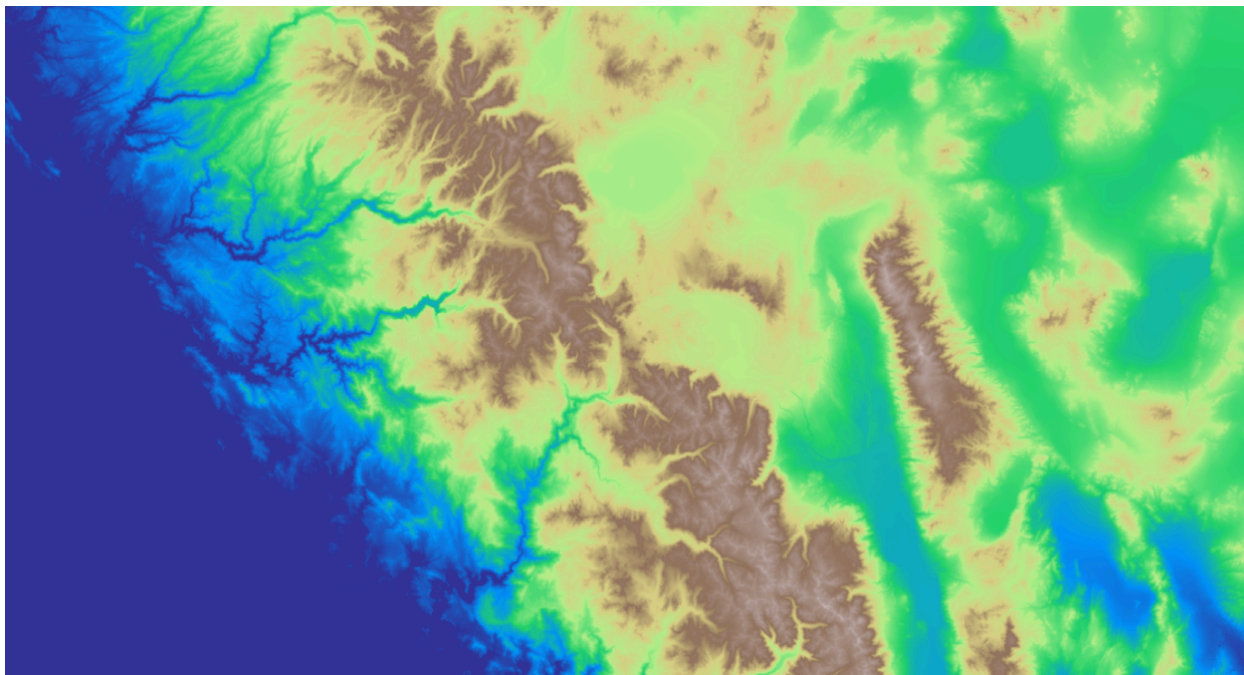


Figure 130: The color relief visualization of the digital elevation model.

The color relief process interpolates colors between the defined elevation breakpoints, creating smooth color transitions that highlight elevation patterns.

26.15.5. Combining Hillshade and Color Relief

The most effective terrain visualizations often combine multiple techniques to show both elevation patterns and surface relief (Figure 131).

```
gdal_calc.py \  
-A color_relief.tif \  
-B hillshade.tif \  
--outfile=shaded_relief.tif \  
--calc="((A.astype(float) * 0.6) + (B.astype(float) * 0.4))" \  
--type=Byte
```

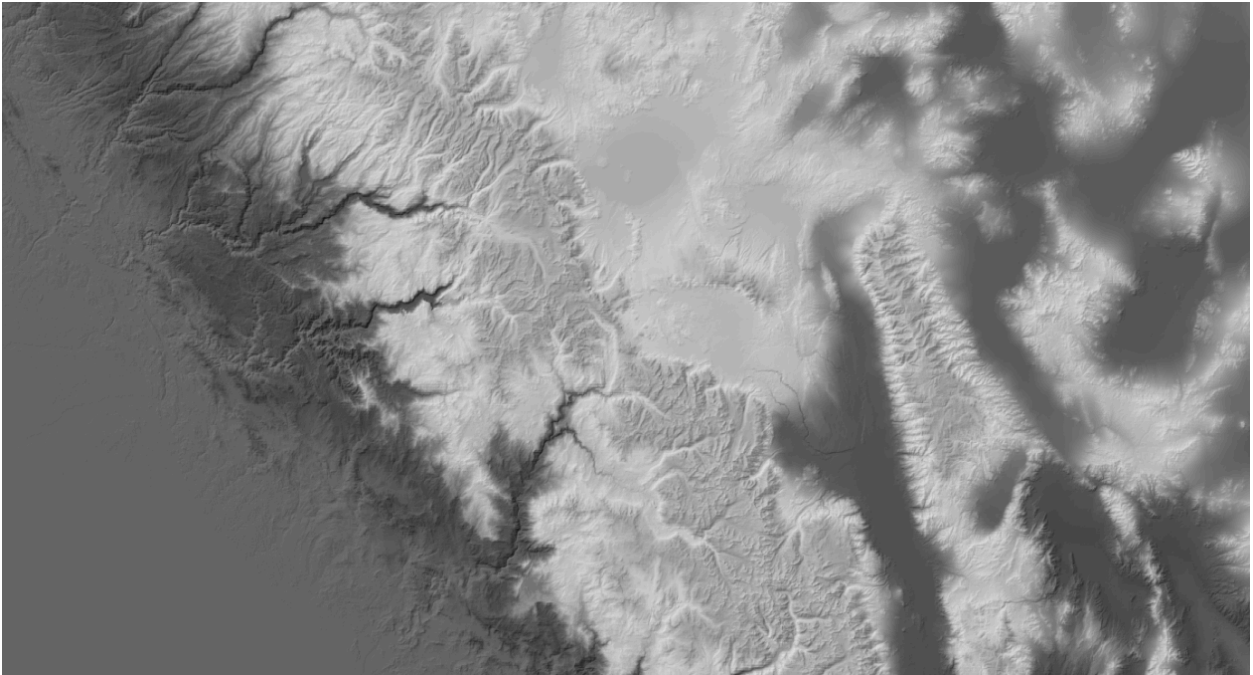


Figure 131: The shaded relief visualization of the digital elevation model.

This blending approach combines 60% color information with 40% hillshading, creating a visualization that shows both elevation patterns and surface texture.

For more sophisticated control over the blending process:

```
gdal_calc.py \
-A color_relief.tif \
-B hillshade.tif \
--outfile=color_hillshade.tif \
--calc="A * (B / 255.0)" \
--type=Byte
```

Since color relief images typically have three bands (RGB), precise blending requires processing each color band separately:

```
gdal_calc.py -A color_relief.tif --A_band=1 -B hillshade.tif --outfile=r.tif --
calc="A*(B/255.0)" --type=Byte
gdal_calc.py -A color_relief.tif --A_band=2 -B hillshade.tif --outfile=g.tif --
calc="A*(B/255.0)" --type=Byte
gdal_calc.py -A color_relief.tif --A_band=3 -B hillshade.tif --outfile=b.tif --
calc="A*(B/255.0)" --type=Byte
```

Then combine the processed bands into a final composite (Figure 132).

```
gdal_merge.py -separate -o color_shaded_relief.tif r.tif g.tif b.tif
```

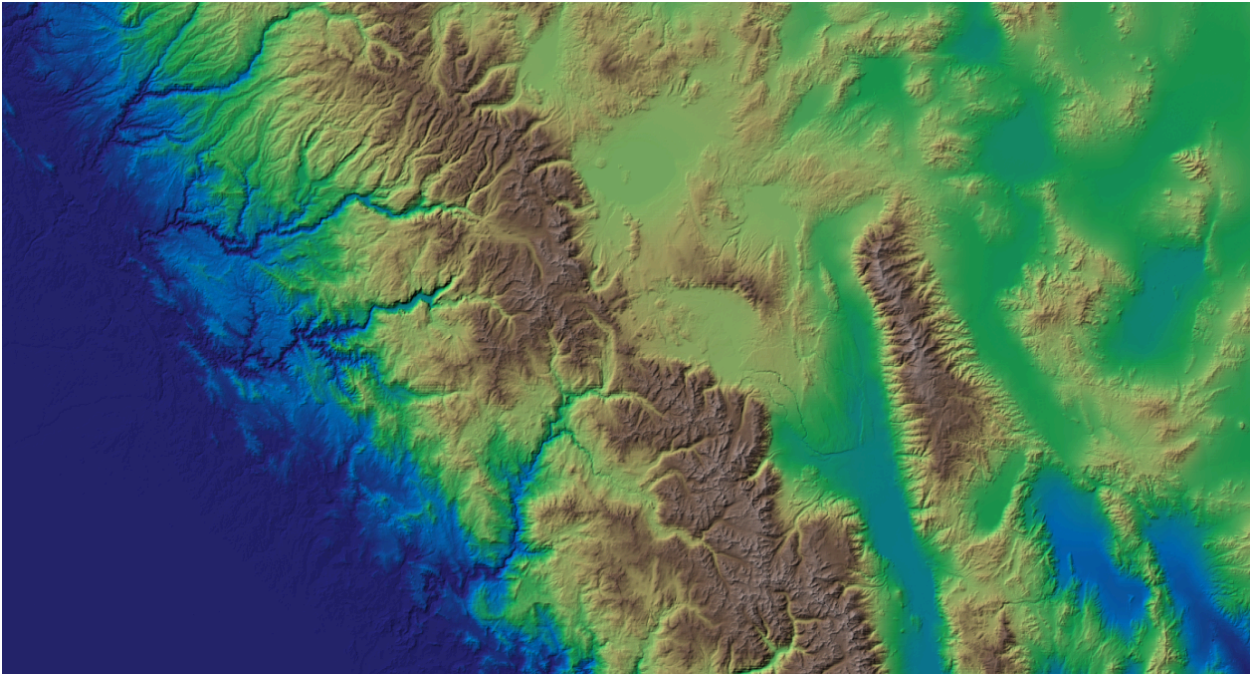



Figure 132: The color shaded relief visualization of the digital elevation model.

This approach provides the highest quality terrain visualization by preserving the full color information while adding realistic shadowing effects.

26.15.6. Generating Contour Lines

Contour lines provide precise elevation information and are essential for many mapping and analysis applications.

```
gdal_contour -i 100 dem.tif contours.shp
```

This command generates contour lines at 100-meter intervals, creating vector features that represent lines of equal elevation ([Figure 133](#)). The interval should be chosen based on the terrain characteristics and the intended use of the contours.

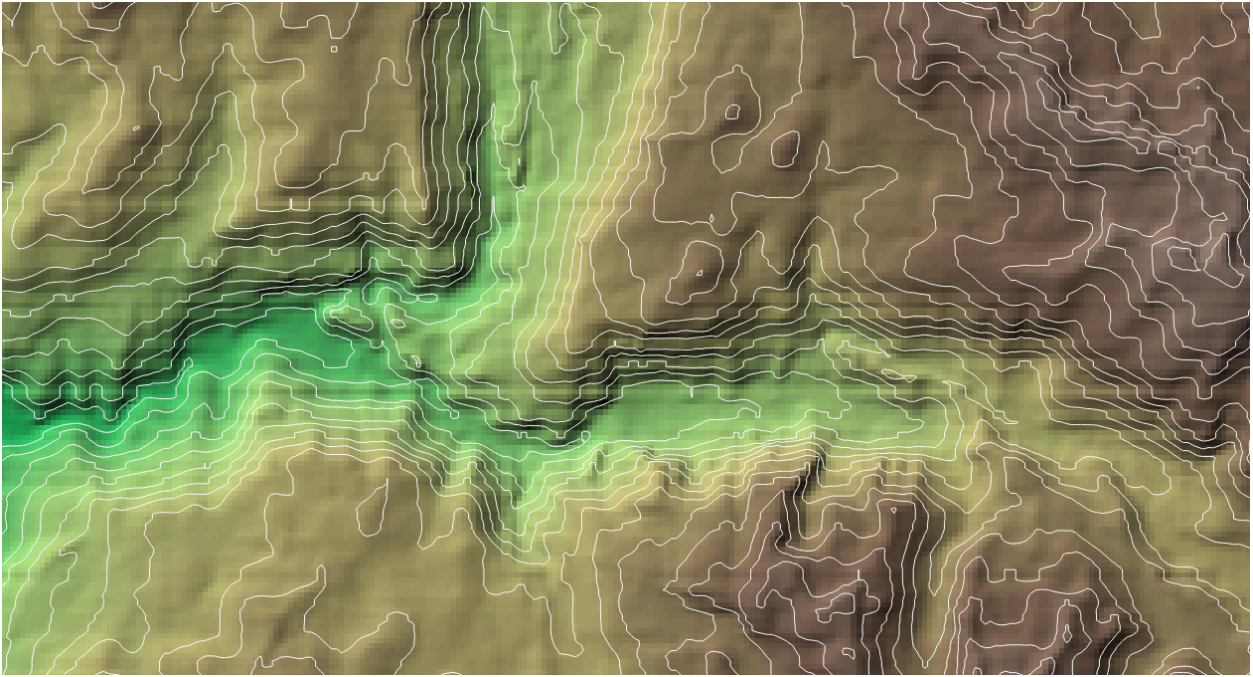


Figure 133: The contour lines of the digital elevation model.

26.16. Key Takeaways

GDAL and OGR provide the fundamental capabilities that underlie most geospatial software and Python libraries. Understanding these tools gives you powerful, direct control over geospatial data and helps you understand what happens behind the scenes in higher-level libraries.

The most important concepts to remember are that GDAL handles raster data while OGR handles vector data, both use a driver-based architecture to support numerous formats, and proper understanding of coordinate systems is crucial when working with these tools. GDAL's coordinate transformation capabilities are essential for working with data from different sources, and its format conversion features make it invaluable for data sharing and interoperability.

The terrain analysis capabilities we explored demonstrate how GDAL can transform basic elevation data into rich information products that support decision-making and analysis. From simple slope calculations to sophisticated visualization products, these tools provide the foundation for understanding landscape patterns and processes.

While libraries like GeoPandas and Rasterio provide more convenient interfaces for most day-to-day tasks, GDAL gives you the power and flexibility to handle any geospatial data challenge. The skills you've learned in this chapter form the foundation for understanding and effectively using the entire Python geospatial ecosystem.

As you continue working with geospatial data, you'll find that understanding GDAL helps you troubleshoot problems, optimize performance, and work with unusual data formats that might not be well-supported by higher-level libraries. The investment in learning these fundamental tools pays dividends throughout your geospatial programming journey.

26.17. References and Further Reading

For an in-depth introduction to GDAL, check out the course on [Mastering GDAL Tools](https://courses.spatialthoughts.com/gdal-tools.html)¹⁴⁰ by [SpatialThoughts](https://spatialthoughts.com)¹⁴¹. This course provides a practical hands-on introduction to GDAL and OGR command-line programs.

For more information about GDAL, please visit the following links:

- [GDAL Documentation](https://gdal.org)¹⁴²
- [GDAL Cheat Sheet](https://github.com/dwtkns/gdal-cheat-sheet)¹⁴³

26.18. Exercises

26.18.1. Exercise 1: Data Inspection and Understanding

Objective: Practice examining geospatial datasets to understand their structure and properties.

Tasks:

1. Use `gdalinfo` to examine the `landsat.tif` file and answer these questions:
 - What is the coordinate reference system?
 - What are the dimensions (width x height) of the image?
 - How many bands does it contain?
 - What is the pixel size/resolution?
2. Use `ogrinfo` to examine the `us_states.geojson` file and determine:
 - How many features (states) are in the dataset?
 - What attributes/fields are available?
 - What is the coordinate reference system?
 - What is the spatial extent (bounding box)?

26.18.2. Exercise 2: Coordinate Transformation Practice

Objective: Practice reprojecting data between different coordinate systems.

Tasks:

1. Reproject the `dem.tif` from its current coordinate system to UTM Zone 11N (EPSG:32611)
2. Reproject the `buildings.geojson` from its current coordinate system to State Plane Nevada West (EPSG:32111)
3. Compare the file sizes before and after transformation - explain any differences you observe

¹⁴⁰<https://courses.spatialthoughts.com/gdal-tools.html>

¹⁴¹<https://spatialthoughts.com>

¹⁴²<https://gdal.org>

¹⁴³<https://github.com/dwtkns/gdal-cheat-sheet>

26.18.3. Exercise 3: Format Conversion and Optimization

Objective: Practice converting between different formats and optimizing data storage.

Tasks:

1. Convert the `buildings.geojson` to three different formats:
 - ESRI Shapefile (.shp)
 - GeoPackage (.gpkg)
 - FlatGeobuf (.fgb)
2. Convert the `dem.tif` to a Cloud Optimized GeoTIFF with DEFLATE compression
3. Compare the file sizes of all formats and discuss the advantages/disadvantages of each

26.18.4. Exercise 4: Spatial Clipping and Analysis

Objective: Practice extracting data for specific areas of interest.

Tasks:

1. Clip the `landsat.tif` using the `las_vegas_bbox.geojson` boundary
2. Clip the `us_states.geojson` to only include states that intersect with a bounding box of your choice

26.18.5. Exercise 5: Raster Band Mathematics

Objective: Practice calculating vegetation indices and raster analysis.

Tasks:

1. Extract the Red (band 4), Near-Infrared (band 5), and Green (band 3) bands from `landsat.tif`
2. Calculate NDVI using the formula: $(\text{NIR} - \text{Red}) / (\text{NIR} + \text{Red})$
3. Create a binary vegetation mask where $\text{NDVI} > 0.4$ represents vegetation
4. Calculate a simple Green-Red ratio: $\text{Green} / \text{Red}$

26.18.6. Exercise 6: Terrain Analysis Workflow

Objective: Practice deriving terrain products from digital elevation data.

Tasks:

1. Using the `dem.tif`, calculate:
 - Slope in degrees
 - Aspect in degrees
 - Hillshade with default lighting
2. Create a multi-directional hillshade for better visualization
3. Generate contour lines at 50-meter intervals

4. Create a custom color relief map with at least 5 elevation classes

26.18.7. Exercise 7: Raster-Vector Conversion

Objective: Practice converting between raster and vector data formats.

Tasks:

1. Create a binary raster from `buildings.geojson` with 1-meter resolution
2. Vectorize the `building_masks.tif` to create polygon features
3. Compare the original building footprints with the vectorized building masks - what differences do you notice?

26.18.8. Exercise 8: Geometry Processing and Data Management

Objective: Practice advanced vector processing operations.

Tasks:

1. Simplify the geometries in `building_masks.gpkg` with a 2-meter tolerance
2. Dissolve the `us_states.geojson` by a common attribute to create a single dissolved polygon for the entire dataset
3. Split the `us_states.geojson` into individual files for each state (save in a 'individual_states' directory)

Chapter 27. Building Interactive Dashboards with Voilà and Solara

27.1. Introduction

Creating interactive web applications from geospatial analyses has traditionally required expertise in web development technologies like HTML, CSS, and JavaScript. However, modern Python frameworks have simplified this process significantly, allowing geospatial analysts to build sophisticated web applications using familiar Python syntax and tools.

This chapter introduces you to two powerful frameworks for creating interactive web applications from your Python geospatial analyses: **Voilà** and **Solara**. These tools bridge the gap between data analysis and web deployment, transforming your Jupyter notebooks and Python scripts into interactive web applications that others can use without needing Python expertise or technical knowledge.

Voilà¹⁴⁴ represents the simplest approach to web application development for geospatial analysts. This framework transforms existing Jupyter notebooks into standalone web applications by removing code cells and preserving only outputs and interactive widgets. The result is a clean, professional interface that allows users to interact with your geospatial data and visualizations without seeing the underlying code. This approach excels for sharing analysis results, creating dashboards, or building simple geospatial tools where the primary goal is to present findings in an accessible format.

Solara¹⁴⁵ takes a more sophisticated approach by providing a reactive web framework specifically designed for data science applications. Built on modern web technologies while maintaining Python-first development, Solara enables you to create complex interactive applications using pure Python code. The framework excels at handling state management and real-time updates, making it particularly well-suited for dynamic geospatial applications that require multi-page interfaces, advanced user interactions, and responsive design elements.

Both frameworks integrate seamlessly with Leafmap's MapLibre backend, enabling you to create stunning 3D globe visualizations, interactive maps with advanced styling, and sophisticated geospatial analysis tools. They also support deployment on cloud platforms like Hugging Face Spaces, enabling you to share your applications with the world at no cost.

The choice between Voilà and Solara often depends on your specific needs and technical requirements. Voilà provides the fastest path from notebook to web application, making it ideal for quickly sharing analysis results or creating simple dashboards. Solara offers more flexibility and control, making it better suited for applications that require complex user interactions, multiple pages, or advanced state management.

While other popular tools like **Streamlit**¹⁴⁶ and **Gradio**¹⁴⁷ exist for creating interactive web applications, they are less suitable for geospatial applications due to limited support for ipywidgets. This limitation prevents them from capturing user interactions with maps, such as drawing polygons or selecting features for analysis. Voilà and Solara, with their native ipywidgets support, can capture these interactions and enable full bidirectional communication between users and geospatial interfaces.

¹⁴⁴<https://voila.readthedocs.io>

¹⁴⁵<https://solara.dev>

¹⁴⁶<https://streamlit.io>

¹⁴⁷<https://www.gradio.app>

Throughout this chapter, we'll explore both frameworks through practical examples, starting with simple globe visualizations and progressing to multi-page applications with interactive sidebars and real-time data visualization capabilities.

27.2. Learning Objectives

By the end of this chapter, you will be able to:

- Understand the differences between Voilà and Solara and when to use each tool
- Install and configure Voilà and Solara with MapLibre support
- Create 3D globe visualizations with Leafmap's MapLibre backend
- Build multi-page Solara applications with navigation
- Implement interactive sidebars with legends, colorbars, and media
- Work with PMTiles for high-performance data visualization
- Create choropleth maps with 3D extrusion effects
- Deploy applications to Hugging Face Spaces for public access
- Apply best practices for creating user-friendly geospatial web interfaces

27.3. Installing Voilà and Solara

Before we can start building web applications, we need to install the necessary packages. Both Voilà and Solara work with Leafmap's MapLibre backend for advanced 3D mapping capabilities.

```
# %pip install voila solara pygis
```

Let's verify that our installations are working correctly by importing the key libraries:

```
import voila
import solara
```

27.4. Introduction to Hugging Face Spaces

[Hugging Face](https://huggingface.co)¹⁴⁸ is a platform for hosting and sharing machine learning models and applications. It is a popular platform for deploying geospatial applications because it is free and easy to use.

To deploy your application to Hugging Face Spaces, you first need to sign up for an account. Navigate to <https://huggingface.co/join> and sign up for an account.

27.4.1. Installing the Hugging Face CLI

To push your application to Hugging Face Spaces, you need to install the `huggingface-cli` command. Uncomment the following cell to install it:

```
# %pip install -U "huggingface_hub[cli]"
```

¹⁴⁸<https://huggingface.co>

27.4.2. Logging in to Hugging Face

To login to Hugging Face, type the following command in your terminal:

```
huggingface-cli login
```

Follow the instructions in the terminal to create a new token at <https://huggingface.co/settings/tokens> and paste it into the terminal. Answer the question with **Y** to confirm that you want to add the token as git credential. Then, you are all set!

27.5. Creating a Basic Voilà Application

27.5.1. Creating a new Hugging Face Space

To create a basic Voilà application, you can duplicate the [Leafmap Voilà](https://huggingface.co/spaces/giswqs/leafmap-voila) space template. Navigate to <https://huggingface.co/spaces/giswqs/leafmap-voila> and click on the three dots in the top right corner and select “Duplicate Space” (see [Figure 134](#)).

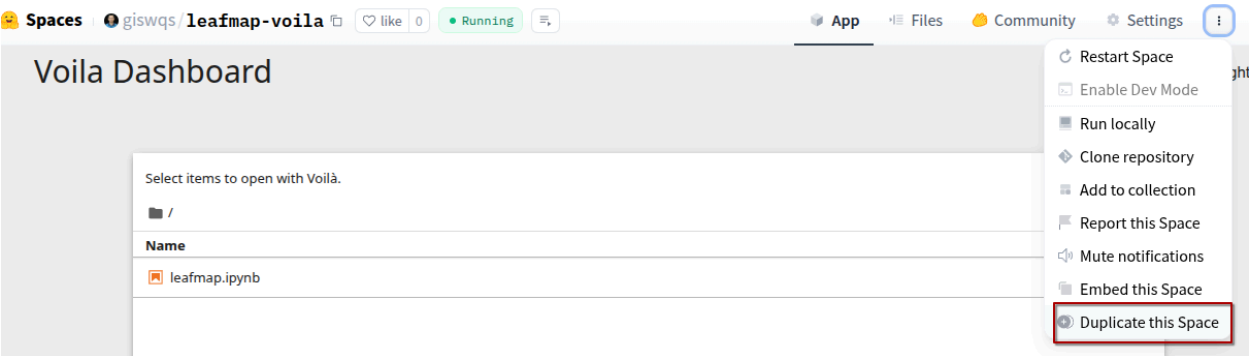


Figure 134: Duplicate the Hugging Face space to your own account.

On the pop-up window, you can change the name of the space if you want. Change the visibility to “Public” (see [Figure 135](#)). Then, click on “Duplicate Space” to create a new space.

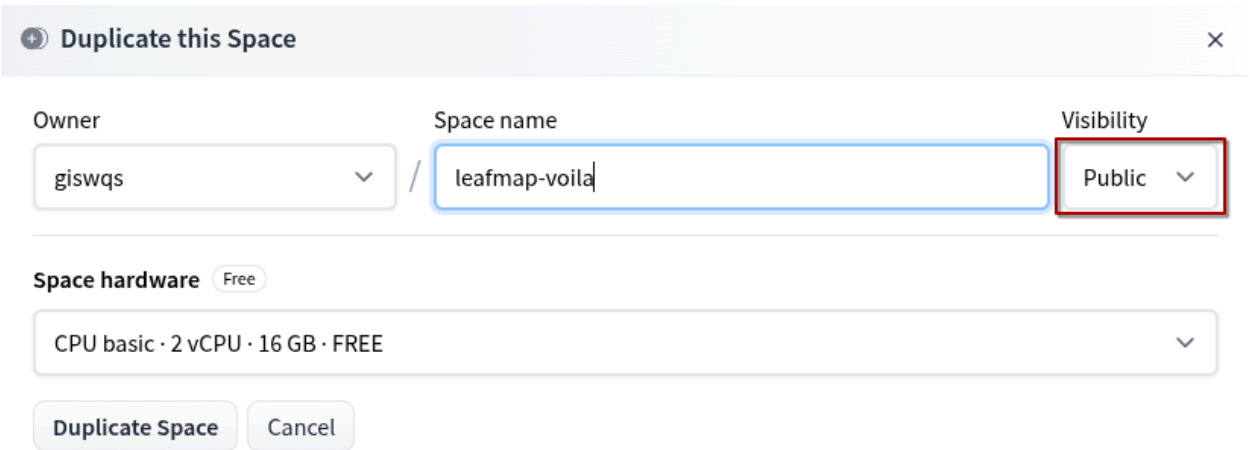


Figure 135: Set the Hugging Face space to public.

It will take a few minutes to create the space. Once the space is created, you can access it at <https://huggingface.co/spaces/YOUR-USERNAME/leafmap-voila>. You may need to refresh the page to see the new space.

27.5.2. Embedding the Hugging Face Space in Your Website

To embed the Hugging Face space in your website, you can click on the three dots in the top right corner of the space and select “Embed Space” (see [Figure 136](#)). The direct link to the space looks like this: <https://giswqs-leafmap-voila.hf.space>. Navigate to the direct link and you will see the space without the Hugging Face menu bar.



Figure 136: Embed the Hugging Face space in your website.

27.5.3. Running the Voilà Application

Once the Voilà Dashboard is running, you can click on the notebook (e.g., `leafmap.ipynb`) to run the application. A 3D globe with a sidebar will be displayed (see [Figure 137](#)).

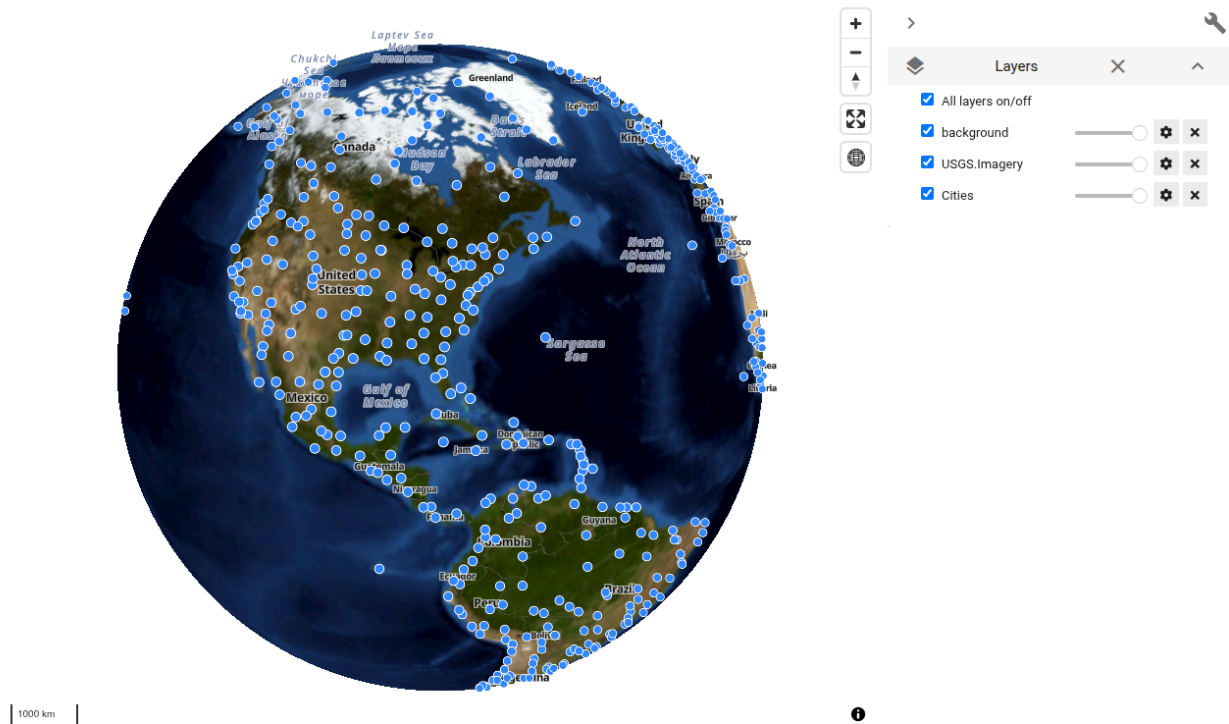


Figure 137: Run the Voilà application.

You can explore the source code of the application in the `leafmap.ipynb` notebook under the `notebooks` folder. The source code like this:

```
import leafmap.maplibregl as leafmap

m = leafmap.Map(
    style="liberty", projection="globe", sidebar_visible=True, height="750px"
)
m.add_basemap("USGS.Imagery")
cities = (
    "https://github.com/opengeos/datasets/releases/download/world/world_cities."
    "geojson"
)
m.add_geojson(cities, name="Cities")
m
```

This is how you would normally use the MapLibre backend in a Jupyter environment. However, Voilà removes the source code from the notebook and only displays the output when the notebook is run, resulting the clean and simple dashboard shown above.

27.5.4. Exploring the File Structure of the Space

Understanding the structure of your Hugging Face space is essential for customizing and maintaining your Voilà application. Let’s examine the key components that make up the space (see [Figure 138](#)).

The space contains several critical files, each serving a specific purpose in the application deployment process. The `README.md` file serves as both the main documentation for your space and contains essential configuration metadata (see [Figure 139](#)). You can update the title and description to reflect your application’s purpose, but the YAML header at the top of the file must remain intact as it contains deployment configuration. The `app_port` specification is particularly important as it tells Hugging Face which port your Voilà application will use.

The `Dockerfile` defines the containerized environment where your application will run. This file specifies the base image, installs dependencies, and sets up the runtime environment. While you typically won’t need to modify this file for basic applications, understanding its role helps you troubleshoot deployment issues if they arise.

The `environment.yml` file manages your application’s Python dependencies using conda. This file ensures that your application has access to all necessary libraries and specific versions required for proper functionality. You can add new packages here as your application grows in complexity, but be mindful of package compatibility and deployment time constraints.

The `notebooks` folder contains the Jupyter notebooks that form the core of your application. Each notebook in this folder becomes accessible through the Voilà interface, allowing users to interact with your geospatial analyses without seeing the underlying code.

main ▾ leafmap-voila

Go to file Ctrl+K

3 contributors

giswqs Update notebook example b7053a7			
github			Add GitHub workflows
notebooks			Update notebook example
.gitignore	Safe	3.44 kB	Initial commit
.pre-commit-config.yaml	Safe	940 Bytes	Add Dockerfile
Dockerfile	Safe	519 Bytes	Update Dockerfile
LICENSE	Safe	1.08 kB	Initial commit
README.md	Safe	575 Bytes	Update prot number
environment.yml	Safe	157 Bytes	Update Dockerfile
voila.json	Safe	178 Bytes	Update Dockerfile

Figure 138: The file structure of the space.

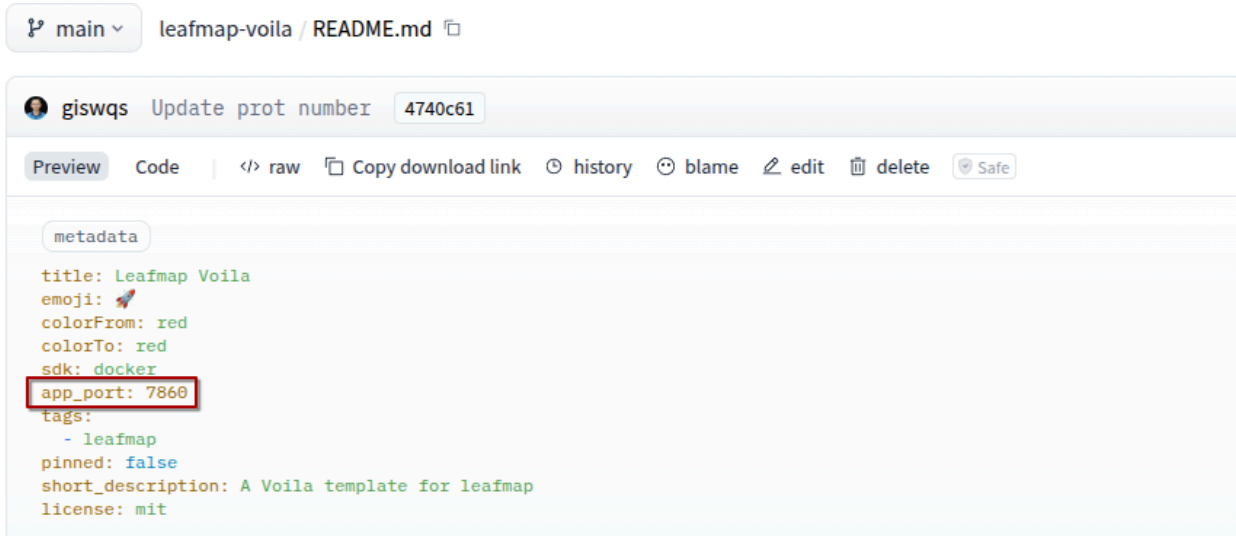


Figure 139: The README file of the space.

27.5.5. Updating the Hugging Face Space

There are two ways to update the Hugging Face space. You can either update the space from the Hugging Face website or update the space from the command line.

27.5.5.1. Updating the Space from the Hugging Face Website

To update existing files on the Hugging Face space, select the “Files” tab in the menu. Then select any file and click on the “Edit” button (see [Figure 140](#)).

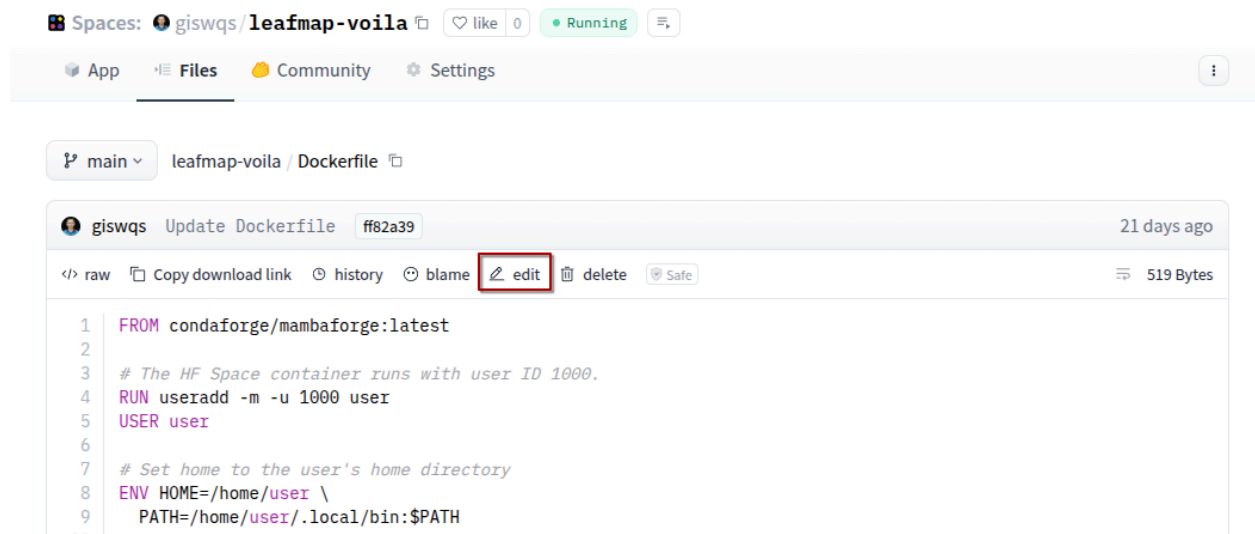


Figure 140: Edit the file on the Hugging Face space.

After editing the file, click on the “Commit changes” button to save the changes.

To add new files, click on the “Contribute” button create a new file or upload existing files (see [Figure 141](#)). Once the files are uploaded, click on the “Commit changes” button to save the changes. The Space will be rebuilt and the changes will be reflected in the space. Wait for a few minutes for the space to be updated. Once the space is updated, you should see the “Running” status next to the space name.

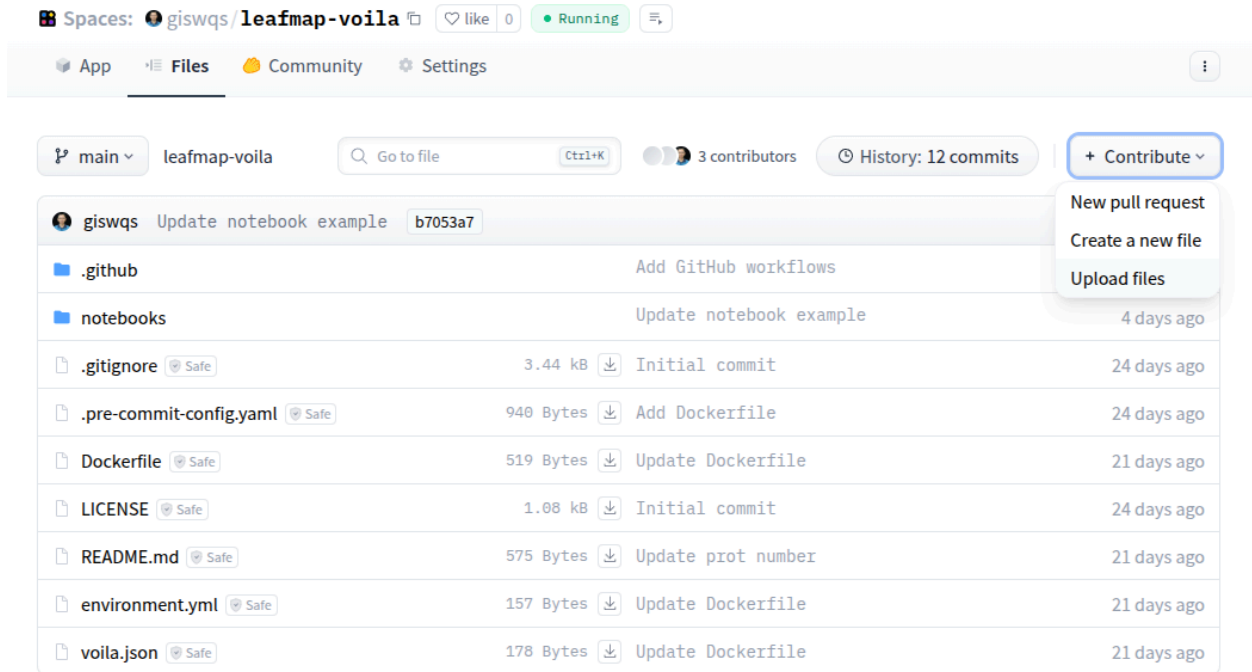


Figure 141: Create a new file on the Hugging Face space.

27.5.5.2. Updating the Space from the Command Line

To update the space from the command line, you first need to clone the repository to your local machine using the following command:

```
git clone https://huggingface.co/spaces/YOUR-USERNAME/leafmap-voila
cd leafmap-voila
```

Then, you can make changes to any file in the space using VS Code. For example, you can update the `README.md` file to change the title and short description of the space. You can modify the dependencies in the `environment.yml` file. You can add or remove notebooks in the `notebooks` folder. To test the application locally, you can run the following command:

```
voila notebooks/
```

A new tab will be opened in your browser with the application running locally. Select a notebook and run it. The notebook will be rendered as a web application. Markdown cells will be rendered as text, and code cells will be rendered as interactive widgets. However, the source code of the notebook will not be rendered. To show the source code in the rendered application, you can stop the running application by pressing `Ctrl+C` in the terminal. Then, run the following command:

```
voila notebooks/ --strip_sources=False
```

The source code of the notebook will be rendered in the application.

You can continue to make changes to the notebook. Once you are satisfied with the changes, you can commit the changes to the repository and push the changes to the Hugging Face space. The space will be rebuilt and the changes will be reflected in the space. Wait for a few minutes for the space to be updated. Once the space is updated, you should see the “Running” status next to the space name.

Congratulations! You have created a Voilà application and deployed it to the Hugging Face space. You can share the application with others by sharing the link to the space.

As you can see, it is very easy to create a Voilà application using your familiar Jupyter notebook with minimal effort. The interactive nature of the notebook makes it easy to explore and visualize geospatial data.

27.6. Creating an Advanced Web Application with Solara

27.6.1. Understanding Solara

Solara represents a significant advancement in Python-based web application development for data science and geospatial analysis. As an open-source library, Solara enables the creation of interactive, data-centric web applications using reusable [UI components](#)¹⁴⁹ that seamlessly integrate with the Python ecosystem you already know and love.

What makes Solara particularly appealing for geospatial professionals is its ability to run applications both inside Jupyter notebooks during development and in production-ready web frameworks like FastAPI, Starlette, and Flask for deployment. This dual-environment capability means you can prototype your geospatial applications in the familiar Jupyter environment and deploy them to the web without major code restructuring.

The framework’s design philosophy centers on scalability and simplicity, built upon trusted technologies and established web standards. Solara enables you to transition effortlessly from quick notebook prototypes to full-featured geospatial data portals. By leveraging Reacton for maintainable code architecture and IPywidgets for rich user interface elements, Solara eliminates the traditional barrier of needing JavaScript or CSS knowledge for web development.

The most significant advantage for geospatial applications comes from Solara’s native support for [ipywidgets](#)¹⁵⁰. This support creates powerful bidirectional communication channels between the frontend interface and backend processing, enabling real-time interactions that are essential for geospatial analysis workflows. Since Leafmap is built upon ipyleaflet and ipywidgets, all of Leafmap’s interactive features integrate seamlessly with Solara applications with minimal code modifications.

This compatibility means that complex geospatial interactions, such as drawing analysis boundaries on maps, selecting features for processing, or adjusting visualization parameters in real-time, work naturally within Solara applications. The framework handles the complex state management and user interface updates automatically, allowing you to focus on the geospatial analysis logic rather than web development concerns.

¹⁴⁹<https://solara.dev/documentation/components>

¹⁵⁰<https://ipywidgets.readthedocs.io>

27.6.2. Using a Leafmap Template for Solara

To streamline the development process, we will use a Leafmap template for Solara. The template is available at <https://huggingface.co/spaces/giswqs/solara-maplibre>. You can duplicate the space to your own account by clicking on the three dots in the top right corner of the space and selecting “Duplicate Space”. In the pop-up window, you can change the name of the space if you want. Change the visibility to “Public”. Then, click on “Duplicate Space” to create a new space. In a few minutes, the space will be created and you can access it at <https://huggingface.co/spaces/YOUR-USERNAME/solara-maplibre>.

Once the space is running, you can access it like this: <https://huggingface.co/spaces/giswqs/solara-maplibre>. It has a home page and a menu bar on the top. The menu bar contains tabs for various pages (Figure 142). Click on any tab to see the content.

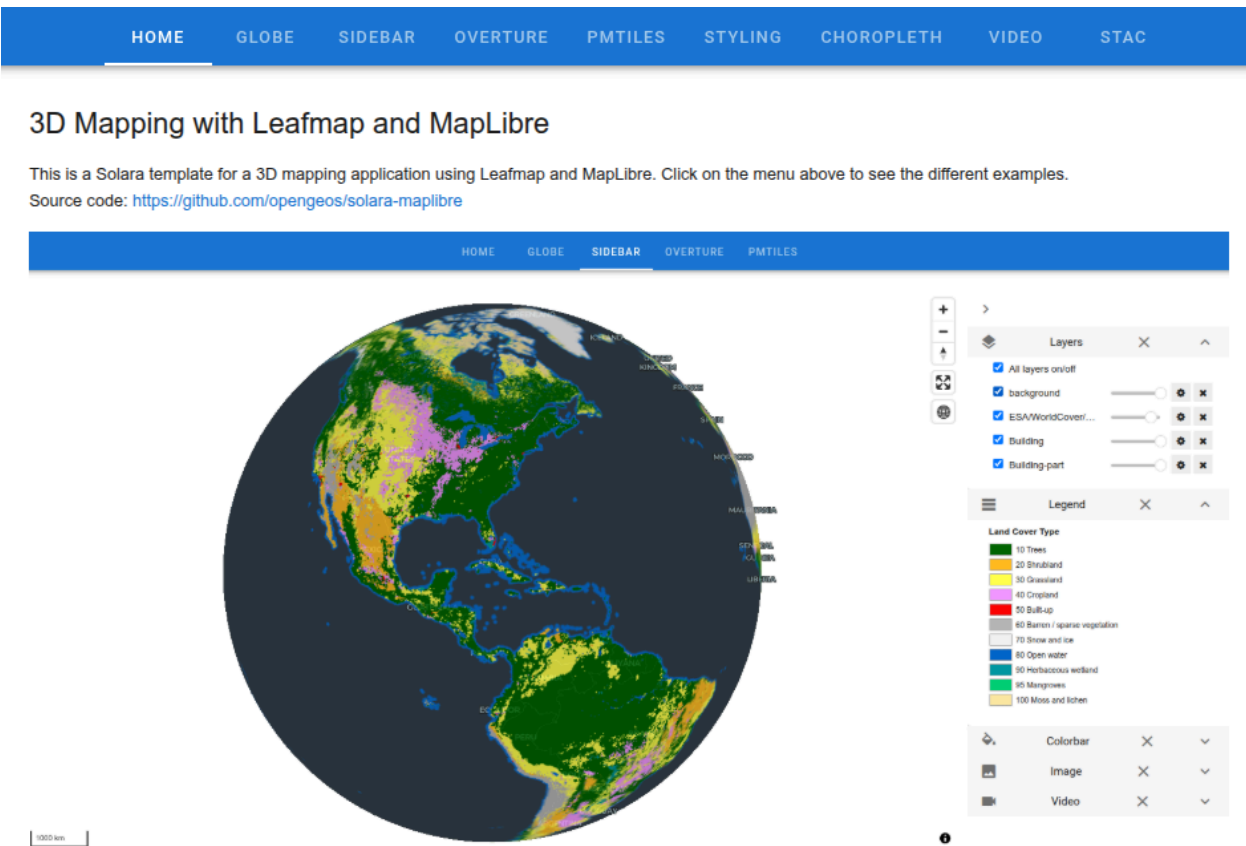


Figure 142: The Solara web app.

Congratulations! You have successfully created a multi-page Solara web app and deployed it to Hugging Face Spaces. It allows you to visualize geospatial data and create stunning 3D visualizations using Leafmap and MapLibre.

27.6.3. Exploring the File Structure of the Solara Web App

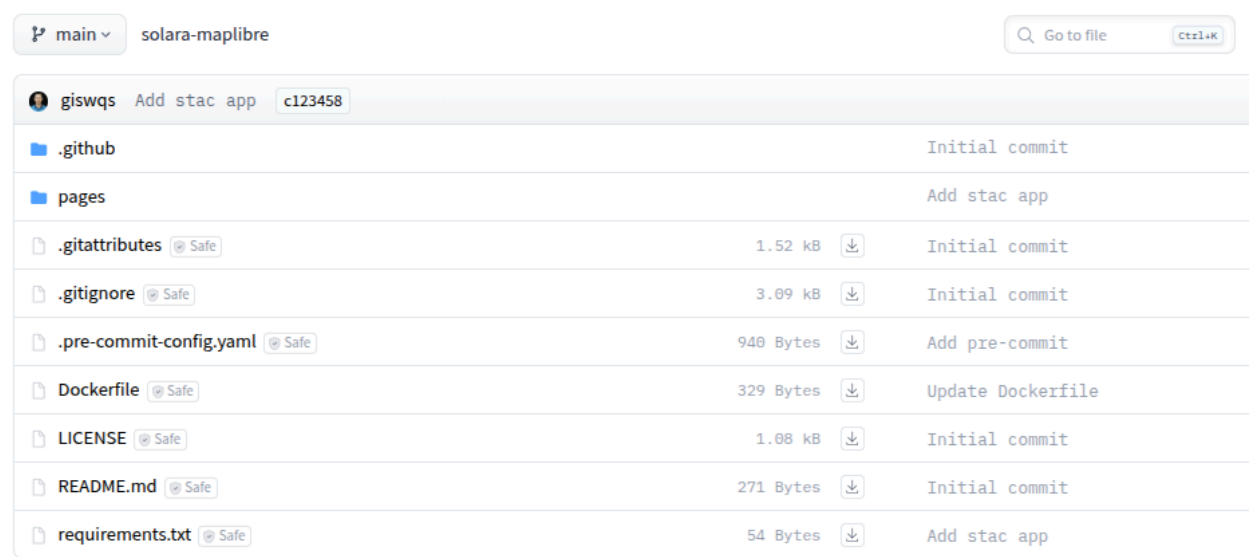
Understanding the organization of a Solara web application is crucial for effective development and maintenance. The file structure follows a logical pattern that separates configuration, dependencies, and application content (see Figure 143).

The `README.md` file functions as both documentation and configuration center for your web application. Similar to the Voilà space, this file contains essential metadata in its YAML header that controls how Hugging Face deploys and runs your application. You can customize the title and description to reflect your application’s purpose, but the header configuration must remain intact to ensure proper deployment.

The `Dockerfile` defines the containerized environment for your application, specifying the base image, dependency installation process, and runtime configuration. While you typically won’t need to modify this file for standard geospatial applications, understanding its structure helps with troubleshooting deployment issues and customizing the runtime environment for specialized requirements.

The `requirements.txt` file manages your Python dependencies using pip, listing all the libraries your application needs to function properly. This file is critical for ensuring reproducible deployments and should include specific version numbers for key dependencies to prevent compatibility issues during deployment.

The `pages` directory represents the heart of your Solara application, containing individual Python scripts that define each page of your multi-page application. Each script in this directory becomes a separate page in your web application’s navigation menu. The naming convention using numerical prefixes (like `01_`, `02_`) determines the order in which pages appear in the navigation, while the descriptive names become the display labels for each tab.



The screenshot shows a GitHub repository interface for 'solara-maplibre'. At the top, there's a search bar with 'Go to file' and a 'Ctrl+K' shortcut. Below the repository name, there's a table listing files and folders. Each row includes a file icon, the filename, a 'Safe' badge, the file size, a download icon, and the commit message. The files listed are: '.github' (Initial commit), 'pages' (Add stac app), '.gitattributes' (1.52 kB, Initial commit), '.gitignore' (3.09 kB, Initial commit), '.pre-commit-config.yaml' (948 Bytes, Add pre-commit), 'Dockerfile' (329 Bytes, Update Dockerfile), 'LICENSE' (1.08 kB, Initial commit), 'README.md' (271 Bytes, Initial commit), and 'requirements.txt' (54 Bytes, Add stac app).

File/Folder	Size	Action
.github		Initial commit
pages		Add stac app
.gitattributes	1.52 kB	Initial commit
.gitignore	3.09 kB	Initial commit
.pre-commit-config.yaml	948 Bytes	Add pre-commit
Dockerfile	329 Bytes	Update Dockerfile
LICENSE	1.08 kB	Initial commit
README.md	271 Bytes	Initial commit
requirements.txt	54 Bytes	Add stac app

Figure 143: The file structure of the Solara web app.

27.6.4. Introduction to Solara Components

Solara components are the building blocks of the web app. They are used to create the UI of the web app. Components can be simple like a button or a text input, or complex like a map or a chart.

To create a component, we can define a Python function decorated with `@solara.component`. The function returns a `solara.Component` object. For example, the following code creates a simple interactive map with a 3D globe. First, we import the Solara and Leafmap libraries. Then, we define a function `create_map` that creates a map with a 3D globe. Finally, we define a component `Page` that uses the `create_map` function to create a map and returns the map as a `solara.Component` object using

`m.to_solara()` . If you run this code in JupyterLab, you will see a map with a 3D globe. This is the same code as the one used by the [globe page](#)¹⁵¹ in the Solara web app you just created.

```
import solara
import leafmap.maplibregl as leafmap

def create_map():

    m = leafmap.Map(
        style="liberty",
        projection="globe",
        height="750px",
        zoom=2.5,
        sidebar_visible=True,
    )
    return m

@solara.component
def Page():
    m = create_map()
    return m.to_solara()

Page()
```

Essentially, you can use leafmap like you normally do in Jupyter notebooks. The only difference is that you need to return the map as a `solara.Component` object using `m.to_solara()` . Place the content within the `Page` component, which is the root component of the web app. Inside JupyterLab, you will need to add `Page()` to the end of the cell to display the map. In a Python script, the `Page` component will be automatically called without the need to add `Page()` to the end of the cell.

Take your time to explore other pages in the Solara web app. You will see that the code structure of each page is very similar to the one we used above.

27.6.5. Creating a New Page

To create a new page, you can create a new Python script in the `pages` folder. The template already contains eight pages. You can remove the existing pages and create new ones. The new page should be named like `01_your-page-name.py` . The number is used to sort the pages. The name should be short and descriptive as it will be used as the tab name in the web app. For example, you can create a new page named `09_mgrs.py` to show a MGRS map for visualizing the Military Grid Reference System (MGRS). Copy the code below and paste it into the new page.

¹⁵¹https://huggingface.co/spaces/giswqs/solara-maplibre/blob/main/pages/01_globe.py

```

import solara
import leafmap.maplibregl as leafmap

def create_map():

    m = leafmap.Map(
        style="positron",
        projection="globe",
        height="750px",
        zoom=2.5,
        sidebar_visible=True,
    )
    geojson = "https://github.com/opegeos/datasets/releases/download/world/mgrs_
grid_zone.geojson"
    m.add_geojson(geojson)
    m.add_labels(geojson, "GZD", text_color="white", min_zoom=2, max_zoom=10)
    return m

@solara.component
def Page():
    m = create_map()
    return m.to_solara()

```

27.6.6. Running the Solara Web App Locally

To run the Solara web app locally, you can use the following command:

```

cd solara-maplibre
solara run pages/

```

This will start the web app on your local machine. You can access it at `http://localhost:8765`. Click on the `MGRS` tab to see the MGRS map you just created ([Figure 144](#)).

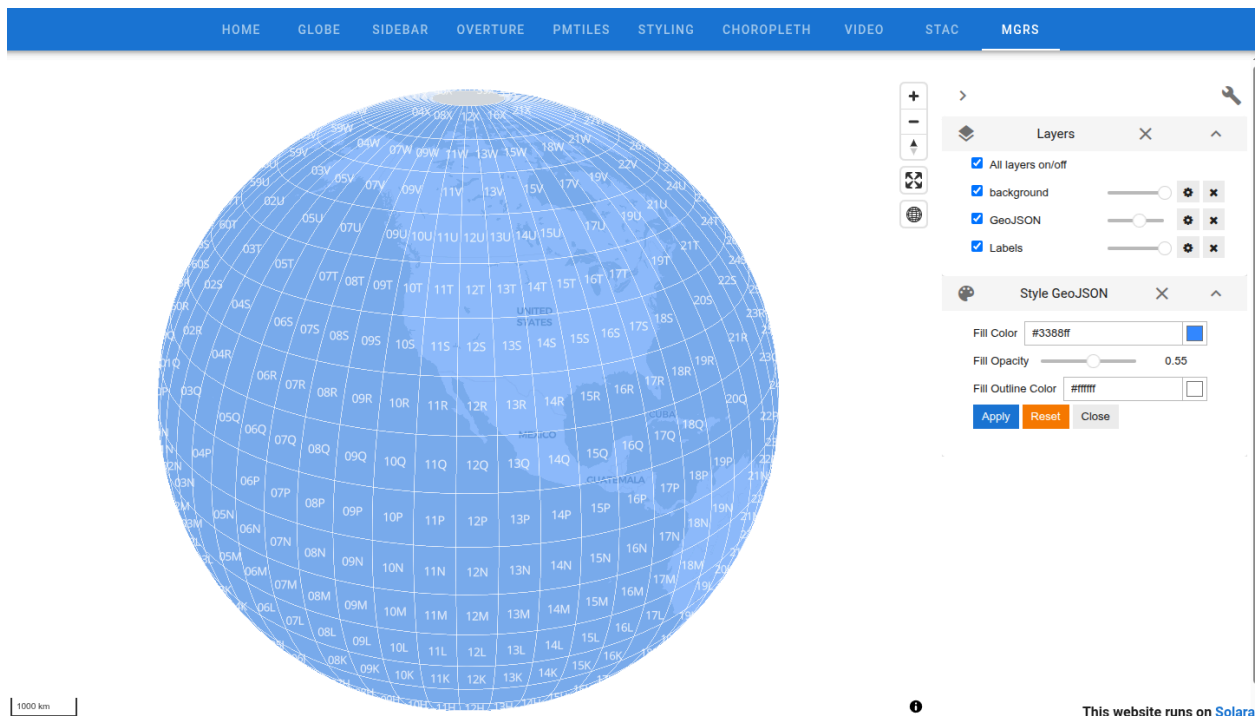


Figure 144: Visualizing the Military Grid Reference System (MGRS) with Solara.

27.6.7. Pushing Changes to the Hugging Face Space

You can continue to update the web app by editing the files in the `pages` folder. Once you are done, you can push the changes to the Hugging Face space using Git within VS Code.

```
git add .
git commit -m "Update the web app"
git push
```

The Hugging Face space will be re-built and the changes will be deployed to the space. Wait for a few minutes until you see the `Running` status in the Hugging Face space. Then, you can access the web app at <https://huggingface.co/spaces/YOUR-USERNAME/solara-maplibre>.

Congratulations! You have successfully created a multi-page Solara web app and deployed it to Hugging Face Spaces. Share your web app with your friends and colleagues.

27.7. Key Takeaways

This chapter introduced you to two powerful frameworks for transforming geospatial analyses into interactive web applications. Understanding when and how to use Voilà and Solara opens up new possibilities for sharing your work and creating accessible geospatial tools.

Voilà provides the simplest path from Jupyter notebook to web application, making it ideal for quickly sharing analysis results or creating straightforward dashboards. Its strength lies in its ability to transform existing notebooks with minimal modification, preserving the familiar development workflow while

creating professional-looking web interfaces. The framework excels when your primary goal is to present findings or enable basic interactions with geospatial data.

Solara offers more sophisticated capabilities for creating complex, multi-page applications with advanced user interactions. Its component-based architecture and reactive programming model make it well-suited for applications requiring state management, real-time updates, and complex user workflows. The framework's native support for ipywidgets ensures that all Leafmap functionality translates seamlessly to web applications.

Both frameworks integrate naturally with the geospatial Python ecosystem, particularly through Leafmap's MapLibre backend, enabling stunning 3D visualizations and interactive mapping capabilities. Their support for deployment on platforms like Hugging Face Spaces makes sharing applications with global audiences both simple and cost-effective.

The choice between frameworks depends on your specific requirements: use Voilà for quick deployment of notebook-based analyses and simple dashboards, and choose Solara for complex applications requiring multiple pages, advanced interactivity, or sophisticated state management. Both frameworks eliminate the traditional barrier of needing web development expertise, allowing geospatial analysts to focus on their domain expertise while creating professional web applications.

27.8. Exercises

27.8.1. Exercise 1: Create a Simple Voilà Dashboard

Create a basic Voilà application that displays a choropleth map of world population data. Start by creating a new Jupyter notebook that loads world [country boundaries and population data](https://github.com/opengeos/datasets/releases/download/vector/countries.geojson)¹⁵², creates a choropleth visualization using Leafmap. Deploy your notebook as a Voilà application to Hugging Face Spaces and share the public URL.

27.8.2. Exercise 2: Build a Multi-Page Solara Application

Develop a multi-page Solara application that showcases different aspects of climate data visualization. Create at least three pages visualizing point, line, and polygon data, respectively. You can use your own data or use the sample data from the [opengeos/datasets](https://github.com/opengeos/datasets/releases/tag/vector)¹⁵³ repository. Deploy your application to Hugging Face Spaces and share the public URL.

¹⁵²<https://github.com/opengeos/datasets/releases/download/vector/countries.geojson>

¹⁵³<https://github.com/opengeos/datasets/releases/tag/vector>

Chapter 28. Distributed Computing with Apache Sedona

28.1. Introduction

In today's data-driven world, geospatial datasets are growing exponentially in both size and complexity. From GPS tracking data and satellite imagery to smart city sensor networks and social media geotagged content, we're generating spatial data at unprecedented rates. Traditional single-machine geospatial processing tools, while excellent for moderate-sized datasets, often struggle with the scale and computational demands of modern big geospatial data applications.

[Apache Sedona](https://sedona.apache.org)¹⁵⁴ (formerly GeoSpark) addresses this challenge by bringing distributed computing capabilities to geospatial data processing. Built on top of [Apache Spark](https://spark.apache.org)¹⁵⁵, Sedona extends Spark's distributed computing framework with comprehensive spatial data types, functions, and algorithms. This combination enables you to process massive geospatial datasets across clusters of machines while maintaining the familiar SQL and DataFrame APIs that make Spark accessible to data scientists and analysts.

Apache Sedona fills a critical gap in the geospatial big data ecosystem. While traditional GIS software excels at interactive analysis and visualization, and databases like PostGIS handle spatial queries efficiently, neither is designed for the massive-scale, distributed processing that modern geospatial applications require. Sedona combines the best of both worlds: the analytical power of distributed computing with the rich spatial functionality that geospatial professionals need.

The power of Sedona lies in its comprehensive approach to distributed geospatial computing. It provides native support for standard spatial data types including Points, LineStrings, Polygons, and complex geometries, all distributed seamlessly across Spark clusters. The framework includes sophisticated spatial operations such as distance calculations, intersections, unions, buffers, and other spatial relationships, all optimized to work at massive scale. Perhaps most importantly, Sedona implements advanced spatial indexing using R-tree and Quad-tree structures, enabling efficient spatial queries on datasets containing billions of spatial objects.

One of Sedona's greatest strengths is its integration with the broader data ecosystem. The platform offers full spatial SQL support through Spark SQL, making it accessible to users comfortable with SQL syntax. It provides APIs in multiple programming languages including Scala, Java, Python, and R, ensuring compatibility with diverse development environments.

Understanding when to use Apache Sedona is crucial for effective implementation. The platform excels when working with datasets that exceed the capacity of traditional GIS tools, typically those involving millions or billions of spatial records. It's particularly valuable for batch processing scenarios where you need to perform complex spatial analysis on massive datasets, such as analyzing years of GPS tracking data or processing satellite imagery archives. Sedona also shines in real-time spatial analytics applications, where streaming spatial data needs immediate processing and analysis. For organizations already using Spark-based data pipelines, Sedona provides a natural extension that adds spatial capabilities without requiring a complete architectural overhaul. Finally, when your analysis requirements demand scaling across multiple machines or cloud clusters, Sedona's distributed architecture provides the necessary infrastructure to handle enterprise-scale geospatial workloads.

¹⁵⁴<https://sedona.apache.org>

¹⁵⁵<https://spark.apache.org>

28.2. Learning Objectives

By the end of this chapter, you will be able to:

- Understand the architecture and benefits of distributed geospatial computing
- Install and configure Apache Sedona with PySpark
- Create and manipulate spatial DataFrames using Sedona’s spatial data types
- Perform fundamental spatial operations like distance calculation, spatial relationships, and geometric transformations
- Execute efficient spatial joins on large datasets using Sedona’s optimized algorithms
- Use spatial indexing to improve query performance on big geospatial data
- Write spatial SQL queries using Sedona’s spatial functions
- Apply Sedona to real-world geospatial analysis scenarios using common datasets

28.3. Installing and Setting Up Apache Sedona

28.3.1. Installation Requirements

Apache Sedona requires Java 8 or higher and is compatible with Apache Spark 3.0 and later versions. The framework’s installation involves several dependencies that need to be properly configured for optimal performance. While there are multiple ways to install and configure Sedona, using a pre-configured Docker container provides the most straightforward path to getting started with distributed geospatial computing.

The Docker approach eliminates many of the common installation challenges associated with Java dependencies, Spark configurations, and spatial library dependencies. The container comes pre-loaded with all necessary components, including the appropriate versions of Spark, Sedona, and supporting geospatial libraries. Use the following command to start a Docker container with Sedona installed:

```
docker run -it -p 8888:8888 -p 4040:4040 -p 8080:8080 -p 8082:8081 -p 7077:7077 -p 8085:8085 -v $(pwd):/app/workspace giswqs/pygis:sedona
```

For cloud-based alternatives, [Wherobots](https://wherobots.com)¹⁵⁶ provides a managed cloud platform specifically designed for Apache Sedona workloads. This managed service handles the infrastructure complexity while providing a familiar Jupyter notebook interface for development and analysis. Navigate to [Wherobots](https://wherobots.com) and sign up for a free account. Once you have an account, you can start a new notebook and select the “Tiny” instance. This will give you free access to a JupyterLab instance with Apache Sedona installed. Within the JupyterLab interface, you can upload the code examples in this chapter to your notebook and run them.

28.3.2. Core Imports and Configuration

Before diving into spatial data processing with Sedona, we need to import the essential libraries and establish our development environment. The process begins with importing the core Sedona modules alongside complementary Python geospatial libraries that enhance our analytical capabilities. This combination provides us with both the distributed computing power of Sedona and the familiar interfaces of traditional Python geospatial tools.

¹⁵⁶<https://wherobots.com>

```
import leafmap
import geopandas as gpd
import pandas as pd
from sedona.spark import *
from pyspark.sql.functions import col, expr
```

28.3.3. Creating a Sedona-Enabled Spark Session

Establishing a Sedona-enabled Spark session is the foundation of distributed geospatial computing. This process involves configuring Spark with specialized JAR files that contain Sedona's spatial functionality and setting up various performance optimizations that enable efficient processing of geospatial data at scale. The configuration process also includes enabling [Kryo serialization](#), which significantly improves the performance of spatial object serialization across the distributed cluster.

The configuration shown below represents a comprehensive setup that supports both vector and raster data processing, includes cloud storage connectivity for accessing remote datasets, and optimizes memory usage for large-scale spatial operations. Each configuration parameter serves a specific purpose in creating an efficient distributed geospatial processing environment.

```
config = (
    SedonaContext.builder()
        .appName("SedonaApp") # Application name for tracking in Spark UI
        .config(
            "spark.serializer", "org.apache.spark.serializer.KryoSerializer"
        ) # Faster serialization
        .config(
            "spark.kryo.registrator",
            "org.apache.sedona.core.serde.SedonaKryoRegistrator"
        ) # Spatial object serialization
        .config(
            "spark.jars.packages",
            "org.apache.sedona:sedona-spark-
shaded-3.5_2.12:1.7.2,org.datasyslab:geotools-wrapper:1.7.2-28.5",
        ) # Core Sedona packages
        .config(
            "spark.jars.repositories",
            "https://artifacts.unidata.ucar.edu/repository/unidata-all",
        ) # Additional repositories
        .config(
            "spark.hadoop.fs.s3a.connection.ssl.enabled", "true"
        ) # Enable SSL for S3 connections
        .config(
            "spark.hadoop.fs.s3a.path.style.access", "true"
        ) # Use path-style access for S3
        .config(
            "spark.hadoop.fs.s3a.impl", "org.apache.hadoop.fs.s3a.S3AFileSystem"
        ) # S3 filesystem implementation
```

```

.config(
    "spark.hadoop.fs.s3a.aws.credentials.provider",
    "org.apache.hadoop.fs.s3a.AnonymousAWSCredentialsProvider",
) # Anonymous access to public S3 buckets
.getOrCreate()
)

# Create the Sedona context which adds spatial capabilities to the Spark session
sedona = SedonaContext.create(config)

```

This configuration demonstrates the essential setup requirements for a production-ready Sedona environment. The Kryo serialization configuration significantly improves the performance of spatial object serialization and deserialization across the distributed cluster. The spatial SQL extensions enable the use of spatial functions directly within SQL queries, making Sedona accessible to users familiar with spatial databases. The memory allocation settings ensure that the system has sufficient resources to handle large spatial datasets efficiently. The S3 configuration enables seamless access to cloud-stored geospatial datasets, which is increasingly important for modern geospatial applications that leverage cloud data repositories.

28.4. Downloading Sample Data

Before we begin our exploration of Sedona's capabilities, we need to download the sample datasets that will be used throughout this chapter. These datasets include world cities data, country boundaries, and other geospatial files that demonstrate various aspects of distributed spatial computing.

```

url = (
    "https://github.com/opengeos/datasets/releases/download/book/sedona-sample-
    data.zip"
)
leafmap.download_file(url)

```

The downloaded zip file will be automatically unzipped and placed in the current working directory.

28.5. Core Concepts and Data Structures

28.5.1. Understanding Spatial DataFrames

The foundation of Apache Sedona's distributed geospatial computing lies in its innovative use of Spark DataFrames enhanced with spatial capabilities. Unlike traditional geospatial tools that work with entire datasets loaded into memory, Sedona represents spatial data as distributed DataFrames where geometry columns contain spatial objects that can be processed across multiple machines simultaneously. This approach enables parallel processing of massive spatial datasets that would be impossible to handle on a single machine.

Each spatial DataFrame in Sedona maintains the familiar tabular structure of regular DataFrames while adding specialized geometry columns that store spatial objects. These geometry columns can contain any type of geospatial data, from simple point locations to complex polygonal boundaries. The distributed

nature of these DataFrames means that operations performed on them are automatically parallelized across the available cluster resources, providing significant performance improvements for large-scale spatial analysis.

28.5.2. Spatial Data Types

Sedona embraces the standard Open Geospatial Consortium (OGC) spatial data types, ensuring compatibility with existing geospatial tools and workflows. Understanding these fundamental data types is crucial for effective spatial analysis, as each type serves specific purposes in representing real-world geographic phenomena.

Point geometries represent discrete locations in space, typically expressed as latitude and longitude coordinates. These are fundamental for representing cities, landmarks, sensor locations, or any phenomena that can be conceptualized as occurring at a specific point. LineString geometries represent sequences of connected points, making them ideal for modeling linear features such as roads, rivers, flight paths, or utility lines. The sequential nature of LineString points allows for accurate representation of curved or complex linear features.

Polygon geometries represent enclosed areas and are essential for modeling boundaries, zones, regions, or any spatial phenomena that occupy a defined area. These might include administrative boundaries, land use areas, or environmental zones. Sedona also supports multi-part geometries including MultiPoint, MultiLineString, and MultiPolygon, which enable representation of complex spatial features that consist of multiple discrete parts, such as island chains or multi-part administrative regions.

28.5.3. Creating Spatial DataFrames

The process of creating spatial DataFrames in Sedona follows a systematic approach that transforms regular tabular data into spatially-aware distributed datasets. This transformation is fundamental to leveraging Sedona's distributed computing capabilities for spatial analysis. The process typically involves creating a standard Spark DataFrame from your source data, then adding geometry columns using Sedona's spatial functions to convert coordinate information into proper spatial objects.

Let's demonstrate this process using city data, building upon the concepts from previous chapters but now applying them within Sedona's distributed computing framework:

This example demonstrates the fundamental process of creating spatial DataFrames from coordinate data. We start with a simple list of city names and coordinates, then transform this into a distributed spatial dataset.

```
# Create sample city data with coordinates for major US cities
cities_data = [
    ("New York", 40.7128, -74.0060),
    ("Los Angeles", 34.0522, -118.2437),
    ("Chicago", 41.8781, -87.6298),
    ("Houston", 29.7604, -95.3698),
    ("Phoenix", 33.4484, -112.0740),
]

# Convert to Spark DataFrame - this creates a distributed dataset
cities_df = sedona.createDataFrame(
```

```

    data=cities_data, schema=["city", "latitude", "longitude"]
)

# Create geometry column using ST_Point function - this transforms coordinates
into spatial objects
cities_spatial = cities_df.withColumn("geometry", expr("ST_Point(longitude,
latitude)"))

# Display the resulting spatial DataFrame
cities_spatial.show(truncate=False)

```

The output shows our DataFrame with a new geometry column containing Point objects. Each row represents a city with both tabular attributes (name, coordinates) and spatial geometry. The geometry column contains Well-Known Binary (WKB) representations of the point geometries, which Sedona uses internally for efficient spatial processing.

28.5.4. Working with Real Geospatial Data

Working with real-world geospatial data involves additional considerations such as coordinate reference systems, complex geometries, and data format conversions. Let's demonstrate this using the NYC boroughs dataset, showing how to integrate traditional geospatial tools with Sedona's distributed processing capabilities.

```

# Load NYC boroughs GeoJSON using GeoPandas for initial processing
boroughs_gdf = gpd.read_file("nybb.geojson")

# Reproject from New York State Plane (EPSG:2263) to Albers Equal Area
(EPSG:5070)
# This ensures accurate area calculations for our later analysis
boroughs_gdf = boroughs_gdf.to_crs("EPSG:5070")
boroughs_gdf

```

This initial step uses GeoPandas to load and reproject the data. We're converting from a local coordinate system (New York State Plane) to a continental projection (Albers Equal Area) that's suitable for accurate area and distance calculations across the broader region.

The next step converts our GeoPandas GeoDataFrame into a Sedona spatial DataFrame. This process involves extracting the geometry information as Well-Known Text (WKT) format, which can be easily transferred to Spark, then reconstructing the spatial objects within the distributed computing environment.

```

# Convert to Pandas DataFrame and extract WKT (Well-Known Text) representation of
geometries
boroughs_pd = pd.DataFrame(boroughs_gdf.drop(columns="geometry"))
boroughs_pd["wkt_geometry"] = boroughs_gdf.geometry.to_wkt()

# Create Spark DataFrame from the Pandas DataFrame

```

```

boroughs_df = sedona.createDataFrame(boroughs_pd)

# Convert WKT strings back to spatial geometry objects using Sedona's
ST_GeomFromWKT function
boroughs_spatial = boroughs_df.withColumn(
    "geometry", expr("ST_GeomFromWKT(wkt_geometry)")
)

# Display the key columns of our spatial DataFrame
boroughs_spatial.select("BoroCode", "BoroName", "geometry").show()

```

This conversion process enables us to work with complex polygon geometries in Sedona's distributed environment. The WKT format serves as an intermediate representation that preserves the complete geometric information while enabling transfer between different spatial computing frameworks. The resulting spatial DataFrame contains the same polygon boundaries as our original data, but now distributed across the Spark cluster for scalable processing.

28.6. Spatial Operations and Functions

Apache Sedona provides a comprehensive library of [spatial functions](#)¹⁵⁷ that form the backbone of distributed geospatial analysis. These functions can be seamlessly integrated into both DataFrame operations and SQL queries, offering flexibility in how you approach spatial analysis problems. The extensive function library covers everything from basic geometric calculations to complex spatial relationships, all optimized for distributed processing across large datasets.

28.6.1. Basic Geometric Properties

Fundamental geometric properties form the foundation of spatial analysis. These calculations provide essential information about the size, shape, and spatial characteristics of geometric objects. In Sedona, these operations are distributed across the cluster, allowing you to calculate properties for millions of geometries simultaneously.

Now let's demonstrate basic geometric property calculations using Sedona's spatial functions. These calculations are performed across the distributed cluster, making them efficient even for datasets with millions of polygons.

```

# Calculate geometric properties for each borough
boroughs_with_metrics = (
    boroughs_spatial.withColumn(
        "area_km2", expr("ROUND(ST_Area(geometry) / 1e6, 2)")
    ) # Convert from m² to km²
    .withColumn(
        "perimeter_km", expr("ROUND(ST_Perimeter(geometry) / 1000, 2)")
    ) # Convert from m to km
    .withColumn("centroid", expr("ST_Centroid(geometry)")) # Calculate geometric
center

```

¹⁵⁷<https://sedona.apache.org/latest/api/sql/Function>

```
)

# Display the calculated metrics
boroughs_with_metrics.select("BoroName", "area_km2", "perimeter_km",
"centroid").show()
```

The results show each borough's area in square kilometers, perimeter in kilometers, and centroid coordinates. These calculations are performed using the projected coordinate system (EPSG:5070), which provides accurate measurements in meters that we then convert to more readable units. The centroid represents the geometric center of each borough, useful for various analytical applications such as distance calculations or label placement in maps.

28.6.2. Distance Calculations

Distance calculations represent one of the most fundamental and frequently used spatial operations in geospatial analysis. These calculations are essential for proximity analysis, spatial clustering, and location-based decision making. In distributed computing environments, distance calculations become particularly powerful as they can be performed simultaneously across massive datasets, enabling analysis that would be computationally prohibitive on single machines.

Understanding coordinate reference systems is crucial for accurate distance calculations. The example below demonstrates the importance of coordinate transformation, converting from geographic coordinates (EPSG:4326) to a projected coordinate system (EPSG:5070) that enables accurate distance measurements in linear units rather than degrees.

The first step in accurate distance calculations is ensuring our data uses an appropriate coordinate reference system. Geographic coordinates (latitude/longitude) measure angles, not distances, so we need to transform our city data to a projected coordinate system that uses linear units.

```
# Transform city coordinates from geographic (EPSG:4326) to projected (EPSG:5070)
coordinate system
# This is essential for accurate distance calculations in linear units (meters)
cities_projected = cities_spatial.withColumn(
    "geometry", expr("ST_Transform(geometry, 'EPSG:4326', 'EPSG:5070')")
)

# View the transformed coordinates
cities_projected.show(truncate=False)
```

The transformed coordinates now represent positions in meters from the projection's origin, enabling accurate distance calculations. You'll notice the coordinate values are much larger, as they represent distances in meters rather than degrees.

Now let's calculate distances from each city to Manhattan's centroid. This demonstrates how to perform distance calculations between different spatial datasets, a common requirement in proximity analysis and location-based services.

```

# Calculate distances from each city to Manhattan's centroid
# First, extract Manhattan's centroid as a reference point
manhattan_centroid = (
    boroughs_spatial.filter(col("BoroName") == "Manhattan")
    .select(expr("ST_Centroid(geometry)").alias("manhattan_center"))
    .collect()[0][0] # collect() brings the result to the driver node
)

# Create a new DataFrame with distance calculations
cities_with_distances = (
    cities_projected.withColumn(
        "manhattan_center",
        expr(
            f"ST_GeomFromWKT('{manhattan_centroid.wkt}')"
        ), # Add Manhattan center as reference
    )
    .withColumn(
        "distance_to_manhattan_meters",
        expr("ST_Distance(geometry, manhattan_center)"), # Calculate distance in
meters
    )
    .withColumn(
        "distance_to_manhattan_km",
        expr(
            "ROUND(ST_Distance(geometry, manhattan_center) / 1000, 2)"
        ), # Convert to kilometers and round
    )
)

# Display results ordered by distance (closest first)
cities_with_distances.select("city", "distance_to_manhattan_km").orderBy(
    "distance_to_manhattan_km"
).show()

```

The results show the distance from each city to Manhattan's geometric center, ordered from closest to farthest. New York City appears first because it contains Manhattan. The distances are calculated using the projected coordinate system, providing accurate measurements in kilometers. This type of analysis is fundamental for proximity-based applications such as delivery routing, market analysis, or service area planning.

28.6.3. Spatial Relationships

Spatial relationships form the core of geospatial analysis, enabling us to understand how different geographic features interact with each other. Sedona provides a comprehensive set of [functions](https://sedona.apache.org/latest/api/sql/Predicate)¹⁵⁸ for testing various spatial relationships between geometries, including containment, intersection, adjacency,

¹⁵⁸<https://sedona.apache.org/latest/api/sql/Predicate>

and proximity. These relationship tests are the building blocks for complex spatial queries and are essential for understanding spatial patterns and connections in your data.

The buffer operation demonstrated below is particularly useful for proximity analysis, creating zones of influence around geographic features. This technique is commonly used in urban planning, environmental analysis, and market research to understand the impact or reach of specific locations or features.

Let's demonstrate buffer analysis by creating a 5-kilometer buffer around Manhattan and identifying which cities fall within this zone. Buffer analysis is essential for proximity studies, impact assessments, and service area analysis.

```
# Create a 5-kilometer buffer around Manhattan's boundary
manhattan_with_buffer = boroughs_spatial.filter(
    col("BoroName") == "Manhattan"
).withColumn(
    "buffer_5km", expr("ST_Buffer(geometry, 5000)") # 5000 meters = 5km buffer
)

# Perform spatial join to find cities within the buffer zone
cities_in_buffer = (
    cities_projected.crossJoin(
        manhattan_with_buffer.select("buffer_5km")
    ) # Cross join to compare all cities with the buffer
    .withColumn(
        "within_buffer", expr("ST_Within(geometry, buffer_5km)")
    ) # Test if city point is within buffer
    .filter(col("within_buffer") == True) # Keep only cities that are within the
    buffer
)

# Display cities that fall within the 5km buffer of Manhattan
cities_in_buffer.select("city").show()
```

The results show which cities fall within a 5-kilometer radius of Manhattan's boundary. This type of analysis is commonly used in urban planning to understand the reach of services, in environmental studies to assess impact zones, and in business analytics to define market areas. The buffer operation creates a polygon that extends 5 kilometers outward from Manhattan's boundary, and the spatial join efficiently identifies all cities that intersect with this zone.

28.7. Spatial Joins and Indexing

Spatial joins represent one of Apache Sedona's most powerful and distinguishing features, enabling efficient analysis of relationships between massive spatial datasets. Unlike traditional database joins that match records based on exact attribute values, spatial joins identify relationships based on geometric properties and spatial proximity. This capability is transformative for large-scale geospatial analysis, allowing you to answer complex questions like "which customers are within 5 kilometers of our stores" or "what is the total population within flood risk zones" across datasets containing millions or billions of records.

The efficiency of Sedona's spatial joins stems from sophisticated spatial indexing algorithms that organize spatial data for rapid retrieval. These indexes, including R-tree and Quad-tree structures, dramatically reduce the computational complexity of spatial queries by eliminating the need to compare every geometry with every other geometry. This optimization is crucial when working with large datasets, as the computational complexity of naive spatial joins increases exponentially with dataset size.

28.7.1. Understanding Spatial Join Types

Spatial joins in Sedona encompass various types of geometric relationships, each serving different analytical purposes. Contains operations identify hierarchical relationships, such as finding all points within administrative boundaries or determining which land parcels contain specific features. Intersects operations are broader, identifying any spatial overlap between features, making them useful for identifying potential conflicts or overlaps in spatial planning. Within operations are the inverse of contains, useful for determining the broader context of point features, such as identifying which administrative region contains each city. Distance-based joins create proximity relationships, essential for market analysis, service area planning, and accessibility studies.

28.7.2. Performing Spatial Joins with World Cities

Demonstrating spatial joins effectively requires working with realistic datasets that showcase the power of distributed processing. The world cities dataset provides an excellent example, containing thousands of cities worldwide with their coordinates and attributes. This dataset size allows us to demonstrate the performance benefits of Sedona's optimized spatial join algorithms while working with data that reflects real-world analytical scenarios.

Let's begin by loading the world cities dataset, which contains 1,249 cities with their geographic coordinates and population information. This dataset will serve as an excellent example for demonstrating spatial joins at scale.

```
world_cities_pd = pd.read_csv("world_cities.csv")
world_cities_pd.head()
```

This dataset contains essential information for each city including name, country, coordinates, and population. The geographic coordinates will be used to create spatial point geometries for each city.

Now we'll convert this Pandas DataFrame to a Sedona spatial DataFrame, creating point geometries from the coordinate columns. This transformation distributes the data across our Spark cluster and enables spatial operations.

```
# Convert Pandas DataFrame to Spark DataFrame for distributed processing
world_cities_df = sedona.createDataFrame(world_cities_pd)

# Create spatial point geometries from longitude and latitude columns
world_cities_spatial = world_cities_df.withColumn(
    "geometry", expr("ST_Point(longitude, latitude)")
)

# Display sample data showing the spatial DataFrame structure
world_cities_spatial.show(10)
```

The resulting DataFrame shows our cities with their attributes and a new geometry column containing point objects. This spatial DataFrame is now distributed across the cluster and ready for large-scale spatial analysis operations.

28.7.3. Spatial Join Example: Cities by Country

This example demonstrates a spatial join operation, one of the most powerful features of distributed spatial computing. We'll create simplified country boundaries and find which cities fall within each country. This type of operation is essential for geographical analysis and administrative data integration.

```
# Create simplified country boundaries for demonstration
# In production, you would load actual country boundary data from sources like
# Natural Earth
sample_countries = [
    (
        "Australia",
        "POLYGON ((112 -44, 112 -10, 154 -10, 154 -44, 112 -44))",
    ), # Simplified Australia bbox
    (
        "UK",
        "POLYGON ((-8.65 49.9, -8.65 60.9, 1.75 60.9, 1.75 49.9, -8.65 49.9))",
    ), # Simplified UK bbox
]

# Create DataFrame with country boundaries
countries_df = sedona.createDataFrame(sample_countries, schema=["country",
"wkkt"])
countries_spatial = countries_df.withColumn("geometry",
expr("ST_GeomFromWKT(wkkt)"))

# Perform spatial join to find cities within country boundaries
# This operation tests each city point against each country polygon
cities_in_countries = world_cities_spatial.alias("cities").join(
    countries_spatial.alias("countries"),
    expr(
        "ST_Within(cities.geometry, countries.geometry)"
    ), # Spatial predicate: city within country
    "inner", # Only keep cities that are within a country boundary
)
cities_in_countries.show()
```

The spatial join operation efficiently identifies which cities fall within each country's boundaries. This is accomplished through Sedona's optimized spatial indexing, which avoids the need to test every city against every country boundary, making the operation scalable even with millions of records.

Now let's summarize the results by counting how many cities fall within each country. This demonstrates how spatial joins can be combined with traditional aggregation operations for comprehensive analysis.


```
# Aggregate cities by country to get summary statistics
city_counts = cities_in_countries.groupBy("countries.country").count()
city_counts.show()
```

This aggregation provides a summary of how many cities from our dataset fall within each country's boundary. This type of analysis is commonly used in demographic studies, market research, and administrative reporting to understand the distribution of features across geographic regions.

28.8. Advanced Spatial Analysis

Advanced spatial analysis techniques in Apache Sedona leverage the platform's distributed computing capabilities to perform complex analytical operations that would be computationally intensive or impossible on traditional single-machine systems. These techniques combine spatial relationships with statistical analysis, enabling sophisticated insights into spatial patterns, clustering, and geographic trends across massive datasets.

28.8.1. Spatial Aggregations

Spatial aggregation operations represent a crucial component of advanced geospatial analysis, enabling the synthesis of detailed spatial data into meaningful summary information. These operations are particularly powerful in distributed environments where you can aggregate millions of spatial features efficiently across cluster resources. Spatial aggregations combine geometric operations with traditional statistical aggregations, creating new insights from the intersection of spatial relationships and attribute data.

Let's explore spatial aggregation by grouping cities from North American countries and calculating both geometric and statistical summaries. This example demonstrates how to combine spatial and non-spatial aggregations in a single operation.

```
# Perform spatial and statistical aggregations on North American cities
country_unions = (
    world_cities_spatial.filter(
        col("country").isin(
            ["USA", "CAN", "MEX"]
        ) # Filter for North American countries
    )
    .groupBy("country")
    .agg(
        expr("ST_Union_Aggr(geometry)").alias(
            "union_geometry"
        ), # Create union of all city points per country
        expr("COUNT(*)").alias("city_count"), # Count cities per country
        expr("CAST(AVG(population) AS INT)").alias(
            "avg_population"
        ), # Calculate average population per country
    )
)
```

```
country_unions.show()
```

This aggregation creates a spatial union of all city points within each country, effectively creating a MultiPoint geometry that represents the spatial distribution of cities. Simultaneously, it calculates statistical summaries like city count and average population. The `ST_Union_Aggr` function is particularly powerful for creating composite geometries from multiple spatial features, useful for visualization and further spatial analysis.

28.8.2. Spatial Clustering Analysis

Spatial clustering analysis reveals patterns in the geographic distribution of phenomena, identifying areas of high concentration or density that may not be apparent from simple mapping or visualization. This type of analysis is fundamental to understanding urban development patterns, identifying population clusters, detecting spatial outliers, and informing location-based decision making. The grid-based approach demonstrated below is particularly effective for large datasets, as it provides a systematic way to partition space and analyze patterns across different scales.

This example demonstrates grid-based spatial clustering analysis, which partitions the Earth's surface into regular grid cells and analyzes the distribution of cities within each cell. This technique is valuable for identifying urban clusters and understanding global settlement patterns.

```
# Create a 2-degree geographic grid for clustering analysis
grid_analysis = (
    world_cities_spatial.filter(
        col("population") > 100000
    ) # Focus on major cities only
    .withColumn(
        "grid_x", expr("floor(longitude / 2) * 2")
    ) # Create 2-degree longitude grid cells
    .withColumn(
        "grid_y", expr("floor(latitude / 2) * 2")
    ) # Create 2-degree latitude grid cells
    .groupBy("grid_x", "grid_y") # Group cities by grid cell
    .agg(
        expr("COUNT(*)").alias("city_count"), # Count cities in each grid cell
        expr("SUM(population)").alias(
            "total_population"
        ), # Sum population in each grid cell
        expr("ST_Centroid(ST_Union_Aggr(geometry))").alias(
            "grid_center"
        ), # Calculate center of city cluster
    )
)

# Identify and display high-density urban clusters
high_density_grids = grid_analysis.filter(
    col("city_count") >= 5
)
```

```
) # Grid cells with 5+ major cities
high_density_grids.orderBy(col("city_count").desc()).show()
```

The results reveal global urban clusters by identifying grid cells with high concentrations of major cities. Each row represents a 2-degree grid cell containing five or more cities with populations exceeding 100,000 people. This analysis helps identify major urban corridors and megalopolitan regions worldwide, such as the Northeast US corridor, European urban clusters, or densely populated areas in Asia.

28.9. Reading Vector Data

Apache Sedona provides comprehensive support for reading various vector data formats, enabling seamless integration with existing geospatial data workflows. The framework supports standard formats including CSV files with coordinate columns, GeoJSON, Shapefiles, GeoPackage, and the increasingly popular GeoParquet format. This flexibility allows you to work with data from diverse sources while leveraging Sedona's distributed processing capabilities.

28.9.1. Reading CSV

CSV files containing coordinate columns are among the most common sources of spatial data. Sedona can efficiently read these files and convert coordinate columns into spatial geometries. This approach is particularly useful for large datasets from sensors, GPS tracking, or geocoded address data.

```
# Read CSV file and create spatial geometries from coordinate columns
cities_df = (
    sedona.read.option("header", True) # CSV has header row
    .format("CSV")
    .load("world_cities.csv")
    .withColumn(
        "geometry", expr("ST_Point(longitude, latitude)")
    ) # Create point geometries from coords
)
cities_df.show()
```

This approach transforms tabular coordinate data into a spatial DataFrame ready for distributed spatial analysis. The `ST_Point` function creates proper spatial geometries from the longitude and latitude columns, enabling all spatial operations and functions.

28.9.2. Reading GeoJSON

GeoJSON files contain both geometry and attribute information in a standardized format. Sedona can read GeoJSON files directly, but the nested structure requires some transformation to create a flat DataFrame suitable for analysis.

```
# Read GeoJSON file - initial structure shows nested format
cables_df = sedona.read.format("geojson").load("cables.geojson")
cables_df.printSchema()
```

The initial read shows the nested structure typical of GeoJSON files. We need to flatten this structure to make it suitable for DataFrame operations and spatial analysis.

```
# Flatten GeoJSON structure for easier analysis
cables_df = (
    sedona.read.format("geojson")
    .load("cables.geojson")
    .selectExpr("explode(features) as features") # Expand the features array
    .select("features.*") # Select all fields from features
    .withColumn("id", expr("properties['id']")) # Extract property fields
    .withColumn("name", expr("properties['name']"))
    .withColumn("color", expr("properties['color']"))
    .drop("properties") # Remove nested properties object
    .drop("type") # Remove type field
)
cables_df.show(5)
```

This transformation creates a flat DataFrame where each row represents a cable feature with its geometry and attributes as separate columns. The geometry column contains the spatial information while the other columns contain the feature attributes.

28.9.3. Reading Shapefiles

Shapefiles remain one of the most widely used geospatial data formats, despite their technical limitations. Sedona provides native support for reading shapefiles directly into spatial DataFrames.

```
# Read shapefile directly into spatial DataFrame
countries_df = sedona.read.format("shapefile").load("countries.shp")
countries_df.show()
```

Sedona automatically handles the multiple files that comprise a shapefile (.shp, .shx, .dbf, etc.) and creates a unified spatial DataFrame with both geometry and attribute information.

28.9.4. Reading GeoPackage

GeoPackage is an open, standards-based format that overcomes many limitations of shapefiles. It can contain multiple layers and supports various data types. When reading GeoPackage files, you need to specify which table/layer to load.

```
# Read specific table from GeoPackage file
countries_df = (
    sedona.read.format("geopackage")
    .option("tableName", "countries") # Specify the table/layer name
    .load("countries.gpkg")
)
countries_df.show()
```

GeoPackage files can contain multiple spatial and non-spatial tables, making them excellent for data distribution and archival purposes.

28.9.5. Reading GeoParquet

GeoParquet combines the efficiency of the Parquet columnar format with spatial data capabilities. This format is particularly well-suited for large datasets and cloud-based workflows due to its compression efficiency and partial read capabilities.

```
# Read GeoParquet files - optimized for large datasets
states_df = sedona.read.format("geoparquet").load("us_states.parquet")
states_df.show()
```

```
# GeoParquet is excellent for point datasets like cities
us_cities_df = sedona.read.format("geoparquet").load("us_cities.parquet")
us_cities_df.show()
```

GeoParquet files load significantly faster than traditional formats and support efficient spatial indexing, making them ideal for big data geospatial applications.

28.9.6. Reading Cloud-Stored Data

One of Sedona's powerful capabilities is direct access to geospatial data stored in cloud repositories. This example demonstrates reading [National Wetlands Inventory \(NWI\)](#)¹⁵⁹ data stored as Parquet files in Amazon S3, showcasing how to work with massive distributed datasets. The dataset is provided by the [US Fish and Wildlife Service](#)¹⁶⁰ in Shapfile and Geodatabase formats, which have been converted to Parquet files for easier distribution and hosted on [Source Cooperative](#)¹⁶¹.

```
# Read individual state wetlands data from S3 (DC wetlands - smaller dataset for demonstration)
url = "s3a://us-west-2.opendata.source.coop/giswqs/nwi/wetlands/DC_Wetlands.parquet"
df = sedona.read.format("parquet").load(url)
```

```
# Display basic information about the dataset
df.show()
```

The raw data contains geometry information in Well-Known Binary (WKB) format. We need to convert this to a format that Sedona can work with for spatial operations.

¹⁵⁹<https://www.fws.gov/program/national-wetlands-inventory/wetlands-mapper>

¹⁶⁰<https://www.fws.gov/>

¹⁶¹<https://source.coop/giswqs/nwi/wetlands>

```
# Convert WKB geometry to readable WKT format for inspection
df_wkt = df.withColumn("geometry", expr("ST_AsText(ST_GeomFromWKB(geometry))"))
df_wkt.show()
```

For analysis of the complete national dataset, we can use wildcard patterns to read all state files simultaneously. This demonstrates Sedona's ability to process truly massive geospatial datasets.

```
# Read all state wetlands data using wildcard pattern (75.8 GB total)
url = "s3a://us-west-2.opendata.source.coop/giswqs/nwi/wetlands/*.parquet"
df = sedona.read.format("parquet").load(url)
```

```
# Count total wetland features across all states
# This operation may take several minutes due to the dataset size
df.count()
```

This example showcases Sedona's scalability - processing 75.8 GB of wetlands data distributed across multiple files and containing millions of polygon features. The distributed nature of Spark and Sedona makes this analysis feasible on cluster resources.

28.10. Visualizing Vector Data

Sedona provides integration with popular visualization libraries to enable interactive exploration of spatial data processed through distributed computing. These visualization capabilities bridge the gap between large-scale spatial analysis and interactive data exploration.

28.10.1. Using KeplerGL

[KeplerGL](https://docs.kepler.gl/docs/keplergl-jupyter)¹⁶² is a powerful web-based visualization tool that excels at displaying large spatial datasets with interactive controls. Sedona's integration allows you to visualize the results of distributed spatial analysis directly in Jupyter notebooks (see [Figure 145](#)).

```
# Configure the initial map view for global data
config = {
    "mapState": {
        "latitude": 20, # Center latitude
        "longitude": 0, # Center longitude (prime meridian)
        "zoom": 1, # Global zoom level
        "pitch": 0, # Map tilt angle
        "bearing": 0, # Map rotation
    },
}
# Create initial map with cities data
m = SedonaKepler.create_map(cities_df, name="Cities", config=config)
```

¹⁶²<https://docs.kepler.gl/docs/keplergl-jupyter>

```
# Add additional layer to the existing map
SedonaKepler.add_df(m, countries_df, name="Countries")
```

```
# Display the interactive map
m
```



Figure 145: Visualization of the world cities and countries using KeplerGL.

The resulting map provides interactive controls for styling, filtering, and exploring the spatial data, making it easy to identify patterns and relationships in large datasets.

28.10.2. Using PyDeck

[PyDeck](https://deckgl.readthedocs.io)¹⁶³ provides 3D visualization capabilities and is particularly effective for displaying spatial data with additional dimensions like elevation or intensity. For example, [Figure 146](#) shows the world countries using PyDeck.

```
# Create 3D geometry visualization for country boundaries
m = SedonaPyDeck.create_geometry_map(countries_df)
m
```

¹⁶³<https://deckgl.readthedocs.io>



Figure 146: Visualization of the world countries using PyDeck.

Figure 147 shows the world cities using PyDeck.

```
# Create scatterplot visualization for cities with custom point size  
m = SedonaPyDeck.create_scatterplot_map(cities_df, radius_min_pixels=3)  
m
```



Figure 147: Visualization of the world cities using PyDeck.

PyDeck visualizations are particularly useful for understanding spatial density patterns and creating compelling presentations of geospatial analysis results.

28.11. Writing Vector Data

After processing spatial data with Sedona, you often need to export the results for use in other applications or for archival purposes. Sedona supports writing to various formats including GeoParquet and GeoJSON, enabling integration with different downstream workflows.

```
# Load the wetlands data for processing
url = "s3a://us-west-2.opendata.source.coop/giswqs/nwi/wetlands/DC_Wetlands.
parquet"
df = sedona.read.format("parquet").load(url)
df.show()
```

```
# Convert WKB geometry to Sedona geometry objects for spatial operations
df = df.withColumn("geometry", expr("ST_GeomFromWKB(geometry)"))
```

Now we can write the processed data to different formats. The repartitioning operation optimizes the output by controlling the number of output files, which is important for performance with large datasets.

```
# Write to GeoParquet format with optimized partitioning
# Repartitioning to 10 partitions creates 10 output files for better performance
df.repartition(10).write.format("geoparquet").mode("overwrite").save("DE_Wetlands")
```

```
# Write to GeoJSON format for compatibility with web applications
# GeoJSON is widely supported but less efficient for large datasets
df.write.format("geojson").mode("overwrite").save("DE_Wetlands_GeoJSON")
```

The choice of output format depends on your downstream requirements. GeoParquet is optimal for large datasets and further analysis, while GeoJSON provides maximum compatibility with web mapping applications and other GIS tools.

28.12. Reading Raster Data

Apache Sedona extends beyond vector data to provide comprehensive support for raster data analysis in distributed environments. This capability enables processing of satellite imagery, climate data, digital elevation models, and other gridded datasets at scale. Sedona's raster support includes reading various formats, performing map algebra operations, and conducting spatial analysis on massive raster datasets.

28.12.1. Reading NetCDF

NetCDF (Network Common Data Form) is widely used in scientific applications, particularly for climate and oceanographic data. These files often contain multi-dimensional arrays with temporal and spatial dimensions, making them ideal for distributed processing.

```
# Load NetCDF file as binary data
df = sedona.read.format("binaryFile").load("wind_global.nc")
```

```
# Extract metadata information to understand the file structure
record_info_df = df.selectExpr("RS_NetCDFInfo(content) as record_info")

# Display metadata about variables, dimensions, and attributes
record_info = record_info_df.first()["record_info"]
print(record_info)
```

The metadata output shows the structure of the NetCDF file, including available variables, dimensions, and coordinate systems. This information is crucial for extracting the specific data layers you need.

```
# Extract the u_wind variable as a raster using lon/lat coordinates
df = df.withColumn("raster", expr("RS_FromNetCDF(content, 'u_wind', 'lon',
'lat')"))
df.show()
```

This transformation converts the NetCDF data into Sedona's raster format, enabling spatial analysis operations on the wind data.

28.12.2. Reading GeoTIFF

GeoTIFF is the most common format for raster geospatial data, widely used for satellite imagery, digital elevation models, and other gridded geographic data. Sedona can read GeoTIFF files and extract their spatial reference information along with the raster data.

```
dem_df = sedona.read.format("binaryFile").load("dem_90m.tif")
dem_df = dem_df.withColumn("raster", expr("RS_FromGeoTiff(content)"))
```

```
dem_df.show()
```

For large raster datasets, it's often useful to tile the data into smaller chunks for distributed processing. This approach improves parallelization and memory management.

```
# Create a temporary view to enable SQL operations on the raster data
dem_df.createOrReplaceTempView("dem_df")
```

```
# Tile the raster into 256x256 pixel chunks for distributed processing
tiles_df = dem_df.selectExpr("RS_TileExplode(raster, 256, 256) as (x, y,
raster)")
```

```
# Display the tiled dataset - each row represents one tile
tiles_df.show()
```

The tiling process creates multiple smaller raster tiles from the original large raster, with each tile containing coordinates (x, y) and the raster data. This approach enables efficient distributed processing of large raster datasets.

28.13. Visualizing Raster Data

Raster visualization in Sedona allows you to create images directly from raster data for immediate visual inspection. This capability is essential for quality control, data exploration, and communicating results.

```
# Convert raster data to image format for visualization
htmlDF = dem_df.selectExpr("RS_AsImage(raster, 1000) as raster_image")
SedonaUtils.display_image(htmlDF)
```

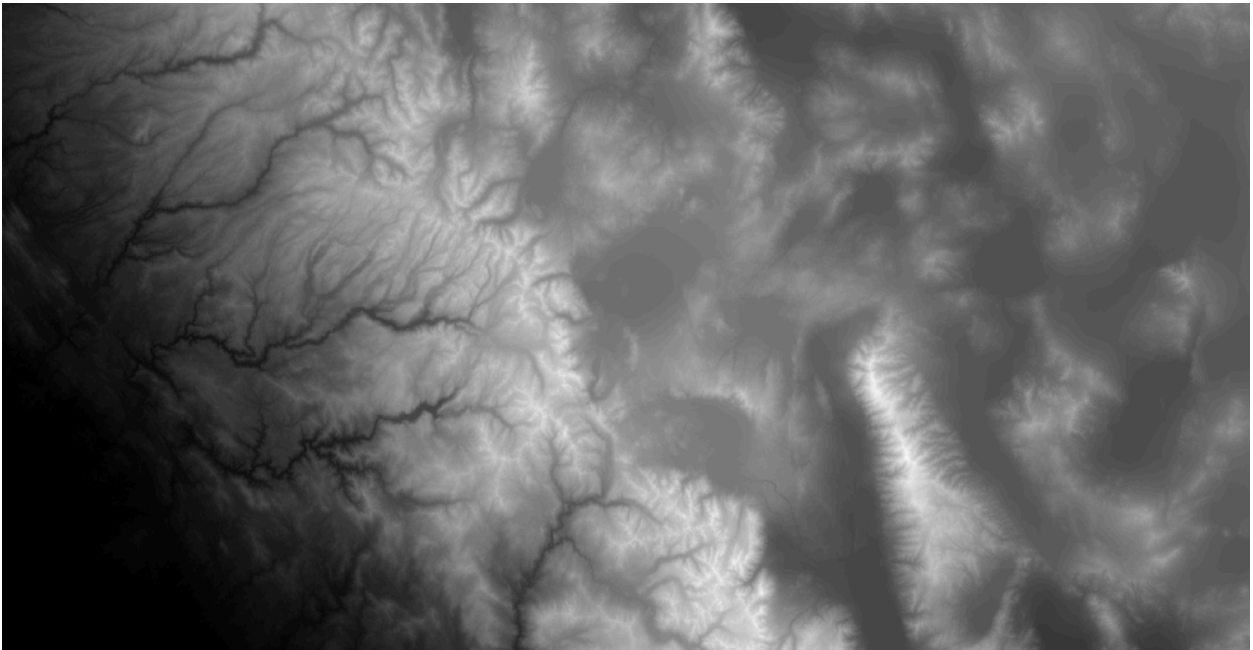


Figure 148: Visualization of the digital elevation model using Sedona.

This visualization ([Figure 148](#)) shows the digital elevation model as a grayscale image where lighter colors represent higher elevations and darker colors represent lower elevations.

28.14. Raster Map Algebra

Map algebra operations enable pixel-level calculations on raster data, essential for creating derived datasets, performing analysis, and implementing spatial models. Sedona's map algebra capabilities support complex mathematical operations on raster pixels.

```
dem_df.printSchema()
```

```
dem_df.createOrReplaceTempView("dem_df")
```

Now we'll create a binary mask identifying mountain areas (elevations above 2000 meters) using map algebra. This type of operation is fundamental for habitat analysis, risk assessment, and land classification.

```
# Create binary mask for areas above 2000m elevation using map algebra
mountain = sedona.sql(
    """
SELECT
    RS_MapAlgebra(
        raster,
        'D',
        'out = (rast[0] > 2000) ? 1 : 0;'
    ) AS mountain
FROM dem_df
    """
)
```

```
# Display the results of the map algebra operation
mountain.show()
```

```
# Visualize the binary mountain mask
htmlDF = mountain.selectExpr("RS_AsImage(mountain, 1000) as raster_image")
SedonaUtils.display_image(htmlDF)
```

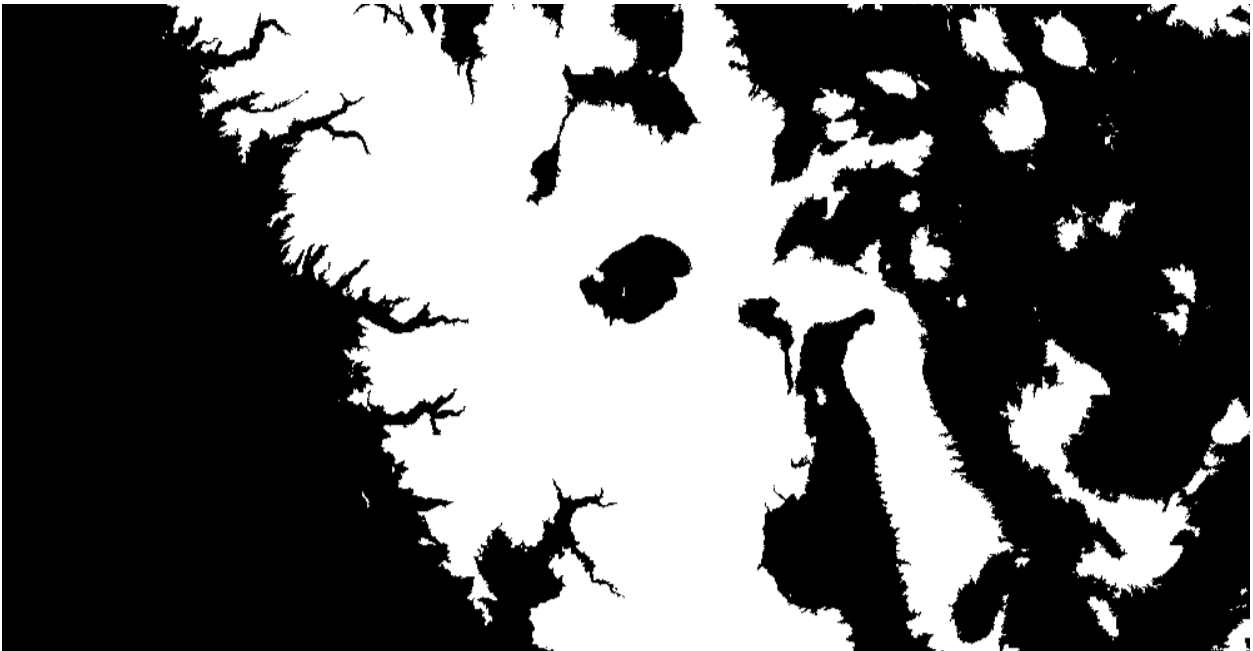


Figure 149: Visualization of the binary mountain mask using Sedona.

The resulting image (Figure 149) shows a binary classification where white pixels represent areas above 2000 meters elevation (mountains) and black pixels represent lower elevations. This type of analysis is fundamental for environmental modeling, risk assessment, and land use planning.

After creating our mountain classification raster, we can save it to disk for future use or sharing with other applications.

```
# Save the mountain classification raster as a GeoTIFF file
mountain.withColumn("raster_binary",
  expr("RS_AsGeoTiff(mountain)")).write.format(
  "raster"
).option("rasterField", "raster_binary").option("fileExtension", ".tiff").mode(
  "overwrite"
).save(
  "mountain"
)
```

This operation saves the binary mountain classification as GeoTIFF files that can be used in other GIS applications or shared with collaborators.

28.15. Raster Zonal Statistics

Zonal statistics enable you to calculate summary statistics for raster values within defined geographic zones or polygons. This is essential for environmental analysis, where you need to understand conditions within specific administrative boundaries, watersheds, or study areas.

```
# Display the elevation raster structure for reference
dem_df.show()
```

Let's define a study area polygon in California's Sierra Nevada mountains to analyze elevation statistics within this specific region.

```
# Define a polygon representing a study area in the Sierra Nevada mountains
polygon = "POLYGON((-119.715885 37.995326, -118.452104 37.995326, -118.452104
37.373679, -119.715885 37.373679, -119.715885 37.995326))"
```

Now we'll calculate the average elevation within this polygon using Sedona's zonal statistics function.

```
# Calculate average elevation within the study area polygon
mount_elev = sedona.sql(
  f"""
select
  RS_ZonalStats(dem_df.raster, ST_Transform(
    ST_GeomFromText('{polygon}'), 'EPSG:4326', 'EPSG:3857'
  ), 1, 'avg', false, false) as avg_elev
from dem_df
```

```
"""  
)
```

```
# Display the calculated average elevation  
mount_elev.show()
```

The result shows the average elevation within our study area polygon. This type of analysis is fundamental for environmental studies, habitat assessment, and regional planning applications.

28.16. Writing Raster Data

Writing processed raster data enables you to save analysis results for future use, share with colleagues, or integrate with other applications. Sedona supports writing raster data in various formats with different compression options for optimal storage and compatibility.

```
# Write the digital elevation model to GeoTIFF with compression  
dem_df.withColumn(  
    "raster_binary", expr("RS_AsGeoTiff(raster, 'LZW', '0.75')")  
)  
.write.format("raster").option("rasterField", "raster_binary").option(  
    "pathField", "path"  
)  
.option(  
    "fileExtension", ".tiff"  
)  
.mode(  
    "overwrite"  
)  
.save(  
    "dem"  
)
```

This example saves the DEM with LZW compression (75% quality) to reduce file size while maintaining data integrity. The compression settings can be adjusted based on your storage and quality requirements.

Let's also demonstrate writing NetCDF data after processing. First, we'll reload and process wind data for demonstration.

```
# Load NetCDF file containing global wind data  
netcdf_df = sedona.read.format("binaryFile").load("wind_global.nc")  
  
# Extract and display metadata to understand the data structure  
record_info_df = netcdf_df.selectExpr("RS_NetCDFInfo(content) as record_info")  
  
# Display metadata information  
record_info = record_info_df.first()["record_info"]  
print(record_info)
```

```
# Extract the u_wind component as a raster for analysis
netcdf_df = netcdf_df.withColumn(
    "raster", expr("RS_FromNetCDF(content, 'u_wind', 'lon', 'lat')")
)
netcdf_df.show()
```

After processing the wind data, you can save it using the same writing operations as shown above, enabling efficient storage and sharing of processed atmospheric data.

28.17. Integration with GeoPandas

One of Sedona's key strengths is its seamless integration with the broader Python geospatial ecosystem. This interoperability enables hybrid workflows where you can leverage Sedona's distributed processing capabilities alongside the rich analysis and visualization features of traditional geospatial tools like GeoPandas. This integration is particularly valuable for workflows that require both large-scale processing and detailed local analysis.

28.17.1. From Sedona DataFrame to GeoPandas GeoDataFrame

Converting Sedona DataFrames to GeoPandas enables local analysis and visualization after distributed processing. This workflow is common when you've processed large datasets with Sedona and want to create detailed visualizations or perform fine-grained analysis on the results.

```
# Load and create spatial data in Sedona format
cities_df = (
    sedona.read.option("header", True)
    .format("CSV")
    .load("world_cities.csv")
    .withColumn("geometry", expr("ST_Point(longitude, latitude)"))
)
cities_df.show()
```

```
# Convert Sedona DataFrame to GeoPandas GeoDataFrame
df = cities_df.toPandas() # Convert to Pandas DataFrame
gdf = gpd.GeoDataFrame(
    df, geometry="geometry"
)
gdf.crs = "EPSG:4326" # Set coordinate reference system
```

```
gdf.explore()
```

This conversion enables you to use all of GeoPandas' visualization and analysis capabilities on data that was processed at scale using Sedona's distributed computing.

28.17.2. From Sedona DataFrame To GeoArrow

GeoArrow¹⁶⁴ provides an efficient interchange format between different geospatial computing libraries, enabling high-performance data transfer with minimal overhead.

```
# Convert Sedona DataFrame to Apache Arrow format for efficient interchange
arrow = dataframe_to_arrow(cities_df)
arrow
```

The Arrow format preserves spatial data types while providing excellent performance for data transfer between different systems and libraries.

28.17.3. From GeoPandas GeoDataFrame to Sedona DataFrame

The reverse conversion enables you to bring data from traditional GeoPandas workflows into Sedona's distributed processing environment. This is useful when you have existing GeoPandas analyses that need to scale to larger datasets.

```
# Load geospatial data using GeoPandas
gdf = gpd.read_file("world_cities.geojson")
gdf.head()
```

```
# Check the coordinate reference system
gdf.crs
```

```
# Convert GeoPandas GeoDataFrame directly to Sedona DataFrame
# Sedona automatically handles the geometry conversion
df = sedona.createDataFrame(gdf)
df.show()
```

The direct conversion from GeoPandas to Sedona preserves both the spatial geometries and attribute data, enabling seamless scaling of existing geospatial workflows.

28.17.4. Advanced Integration: Converting Through GeoArrow

For more complex workflows involving multiple data processing libraries, GeoArrow provides an efficient intermediate format that preserves spatial data types across different systems.

```
# Convert Sedona DataFrame to Arrow format
arrow = dataframe_to_arrow(df)
```

```
# Convert Arrow format back to GeoPandas
gdf = gpd.GeoDataFrame.from_arrow(arrow)
```

¹⁶⁴<https://geoarrow.org>


```
# Display the reconstructed GeoPandas DataFrame
gdf.head()
```

```
# Convert back to Sedona for further distributed processing
df = sedona.createDataFrame(gdf)
df.show()
```

This round-trip conversion demonstrates the interoperability between different geospatial computing environments, enabling flexible workflow design.

28.17.5. Custom Conversion Functions

For specialized workflows, you can create custom conversion functions that handle specific data transformations or optimizations.

```
# Custom function to convert Sedona DataFrame to GeoPandas with error handling
def sedona_to_geopandas(sedona_df, geometry_col="geometry"):
    """Convert Sedona DataFrame to GeoPandas DataFrame with robust error
    handling"""
    # Convert spatial geometries to WKT format for transfer
    with_wkt = sedona_df.withColumn("wkt_geom",
    expr(f"ST_AsText({geometry_col})"))

    # Convert to Pandas DataFrame
    pandas_df = with_wkt.toPandas()

    # Convert WKT strings to Shapely geometries
    from shapely import wkt

    pandas_df["geometry"] = pandas_df["wkt_geom"].apply(wkt.loads)

    # Create GeoPandas GeoDataFrame
    gdf = gpd.GeoDataFrame(pandas_df.drop("wkt_geom", axis=1),
    geometry="geometry")

    return gdf

# Example usage: convert filtered dataset to GeoPandas
major_cities = world_cities_spatial.filter(col("population") > 1000000)
major_cities_gdf = sedona_to_geopandas(major_cities)

print(f"Converted {len(major_cities_gdf)} major cities to GeoPandas")
```

```
# Display the converted major cities dataset
major_cities_gdf
```

This custom function provides a robust way to convert Sedona results to GeoPandas format, enabling detailed analysis and visualization of distributed processing results.

28.18. Real-World Use Cases

These examples demonstrate practical applications of Sedona's distributed spatial computing capabilities for solving real-world analytical challenges. The use cases showcase how Sedona's scalability enables analysis that would be impractical with traditional single-machine approaches.

28.18.1. Use Case 1: Global Urban Density Analysis

This analysis examines urban density patterns across the world's major cities, demonstrating how Sedona can process thousands of cities simultaneously to identify global urban development patterns.

```
# Analyze urban density patterns for cities with population > 500,000
urban_analysis = (
    world_cities_spatial.filter(col("population") > 500000) # Focus on major
cities
    .withColumn(
        "urban_density",
        col("population")
        / expr("ST_Area(ST_Buffer(geometry, 10000))"), # People per m² in 10km
radius
    )
    .withColumn(
        "city_category",
        expr(
            """
CASE
    WHEN population > 5000000 THEN 'Megacity'           -- Cities > 5M people
    WHEN population > 1000000 THEN 'Large City'         -- Cities 1-5M people
    ELSE 'Medium City'                                   -- Cities 500K-1M
people
END
            """,
        ),
    )
)

# Calculate average density by city category
urban_analysis.groupBy("city_category").agg(
    expr("count(*)").alias("count"), # Number of cities in each category
    expr("avg(urban_density)").alias("avg_density"), # Average density per
```

```
category
).show()
```

This analysis reveals global patterns in urban density, showing how different city sizes relate to population concentration. Such analysis is valuable for urban planning, infrastructure development, and comparative urban studies.

28.18.2. Use Case 2: Transit Accessibility Assessment

This use case demonstrates how to assess public transit coverage potential across major urban areas, calculating accessibility metrics that are essential for transportation planning and urban development.

```
# Analyze potential transit coverage for cities with population > 100,000
transit_analysis = (
    world_cities_spatial.filter(col("population") > 100000)
    .withColumn(
        "transit_coverage",
        expr("ST_Buffer(geometry, 1000)"), # 1km radius transit catchment area
    )
    .withColumn(
        "coverage_area_km2",
        expr("ST_Area(ST_Buffer(geometry, 1000)) / 1000000"), # Convert m² to
km²
    )
)

# Calculate global transit accessibility metrics
accessibility_metrics = transit_analysis.agg(
    expr("avg(coverage_area_km2)").alias(
        "avg_coverage_area"
    ), # Average coverage area per city
    expr("sum(coverage_area_km2)").alias(
        "total_coverage_area"
    ), # Total potential coverage globally
)

accessibility_metrics.show()
```

This analysis provides insights into transit accessibility patterns globally, helping transportation planners understand coverage requirements and potential service areas. The 1-kilometer buffer represents typical walking distance to transit stops, making this analysis directly applicable to real-world planning scenarios.

28.19. Key Takeaways

Apache Sedona represents a transformative advancement in geospatial big data processing, fundamentally changing how we approach large-scale spatial analysis. The platform's greatest strength lies in its scalability, enabling the processing of geospatial datasets that would be computationally impossible to handle with traditional single-machine tools. By leveraging Spark's distributed computing framework,

Sedona allows analysts to perform complex spatial operations on billions of spatial objects across multiple machines, breaking through the computational barriers that previously limited geospatial analysis.

The performance improvements offered by Sedona are substantial and multifaceted. The combination of sophisticated spatial indexing algorithms, optimized spatial joins, and distributed processing architecture provides dramatic speed improvements for large-scale spatial analysis tasks. These optimizations become increasingly important as dataset sizes grow, where traditional approaches would become prohibitively slow or impossible to execute.

Integration capabilities make Sedona particularly valuable in modern data ecosystems. The platform seamlessly integrates with the existing Spark ecosystem, enabling organizations to incorporate spatial analysis into their established data pipelines and workflows without requiring complete architectural overhauls. This integration extends beyond technical compatibility to include operational workflows, team skills, and existing infrastructure investments.

Accessibility represents another crucial advantage of Sedona's design. The platform's SQL interface makes advanced spatial analysis accessible to users familiar with database technologies, democratizing access to sophisticated geospatial analysis capabilities. Simultaneously, the DataFrame API provides a familiar interface for data scientists already working with Spark, reducing the learning curve for adopting distributed spatial analysis.

The ecosystem compatibility of Sedona enables hybrid analytical workflows that leverage the strengths of both distributed and traditional geospatial tools. Easy integration with popular Python geospatial libraries like GeoPandas allows analysts to combine the computational power of distributed processing with the rich visualization and analysis capabilities of traditional desktop tools, creating flexible analytical workflows that can adapt to different scales and requirements.

For organizations and analysts working with large geospatial datasets, Apache Sedona provides the essential tools needed to perform complex spatial analysis at scale. This capability makes it an indispensable component in the modern geospatial data scientist's toolkit, particularly as spatial data volumes continue to grow exponentially across industries and applications.

28.20. References and Further Reading

For an excellent video introduction to Apache Sedona and scalable spatial analytics with Spark, watch [Apache Sedona Tutorial for Data Engineers: Scalable Spatial Analytics with Spark](#)¹⁶⁵ by Matt Forrest. This video provides a clear and practical overview of how Sedona integrates with Spark for large-scale geospatial processing.

For more information about Apache Sedona, please visit the following links:

- [Apache Sedona](#)¹⁶⁶
- [Apache Sedona API Documentation](#)¹⁶⁷
- [Cloud Native Geospatial Analytics with Apache Sedona](#)¹⁶⁸

¹⁶⁵https://youtu.be/V__Lq72ge5A

¹⁶⁶<https://sedona.apache.org>

¹⁶⁷<https://sedona.apache.org/latest/api/sql/Overview/>

¹⁶⁸<https://resources.wherobots.com/cloud-native-geospatial-analytics-with-apache-sedona>

28.21. Exercises

28.21.1. Exercise 1: Setting Up Sedona and Basic Spatial Operations

1. **Install and configure Apache Sedona** with PySpark in your environment.
2. **Create a Sedona-enabled Spark session** with appropriate memory configuration.
3. **Create a spatial DataFrame** from the following city coordinates:
 - San Francisco: (37.7749, -122.4194)
 - Seattle: (47.6062, -122.3321)
 - Portland: (45.5152, -122.6784)
 - Los Angeles: (34.0522, -118.2437)
4. **Calculate the area of the convex hull** that encompasses all four cities.

28.21.2. Exercise 2: Working with Real Geospatial Data

1. **Load the NYC boroughs GeoJSON data** using GeoPandas: <https://github.com/opengeos/datasets/releases/download/vector/nybb.geojson>
2. **Convert the data to a Sedona spatial DataFrame** with proper geometry columns.
3. **Calculate the following metrics** for each borough:
 - Area in square kilometers
 - Perimeter in kilometers
 - Centroid coordinates
4. **Rank the boroughs** by area from largest to smallest.

28.21.3. Exercise 3: Distance Analysis

1. **Load the world cities dataset:** https://github.com/opengeos/datasets/releases/download/world/world_cities.csv
2. **Create spatial points** for all cities with population > 1,000,000.
3. **Find the 5 closest major cities** to New York City (40.7128, -74.0060).
4. **Calculate the total population** within 500km of New York City.

28.21.4. Exercise 4: Spatial Joins

1. **Create a simplified country boundary** for the United States using a bounding box:
 - West: -125°, East: -66°, South: 25°, North: 49°
2. **Perform a spatial join** to find all cities from the world cities dataset that fall within this bounding box.
3. **Calculate summary statistics:**
 - Total number of cities
 - Average population
 - Largest city by population

28.21.5. Exercise 5: Spatial Aggregation and Clustering

1. **Load the world cities dataset** and filter for cities with population > 50,000.
2. **Create a 5-degree grid** over the world (divide longitude and latitude by 5 and round).
3. **Group cities by grid cell** and calculate:
 - Number of cities per grid cell
 - Total population per grid cell
 - Average population per grid cell
4. **Find the top 10 most populous grid cells** and display their coordinates and statistics.

28.21.6. Exercise 6: Buffer Analysis

1. **Create 50km buffers** around all major cities (population > 1,000,000) in the world cities dataset.
2. **Find buffer intersections** - identify pairs of cities whose buffers overlap.
3. **Calculate the intersection area** for overlapping buffers.
4. **List the top 5 pairs of cities** with the largest buffer intersection areas.

28.21.7. Exercise 7: Spatial SQL Queries

1. **Register your world cities spatial DataFrame** as a temporary SQL table.
2. **Write SQL queries** to:
 - Find all cities within 100km of London (51.5074, -0.1278)
 - Calculate the density (population per square km) for cities with buffers of 25km radius
 - Find the northernmost and southernmost cities in each continent
3. **Execute the queries** and display the results.

28.21.8. Exercise 8: Advanced Spatial Analysis

1. **Load both the world cities dataset and the NYC boroughs dataset.**
2. **Create a multi-scale analysis:**
 - Find all world cities within 500km of New York City
 - For each NYC borough, count how many world cities fall within 100km of its centroid
 - Calculate the “international connectivity score” as the sum of populations of nearby international cities
3. **Rank the NYC boroughs** by their international connectivity score.
4. **Export the results** to a GeoJSON file for visualization in a web mapping application.

Introduction to GIS Programming

Unlock the Power of Python for Geospatial Analysis and Visualization

This book offers a comprehensive, hands-on introduction to the world of geospatial analysis using open-source Python packages. Designed for learners of all levels, this book breaks down the complexities of Geographic Information Systems (GIS) into clear, actionable steps, making it ideal for students, researchers, professionals, and self-learners interested in mastering spatial data programming.

What sets this book apart is its step-by-step, example-driven approach. Beginning with foundational Python programming skills, you'll build your understanding gradually, progressing to advanced techniques in geospatial analysis. The content is designed to be interactive, with real-world datasets and practical exercises that allow you to apply your skills immediately. You'll work through a variety of projects, from basic spatial data manipulation to building interactive dashboards and cloud-based geospatial applications. Whether you're looking to automate GIS workflows, develop geospatial web applications, or deepen your spatial data science skills, this book will guide you through the entire process with clarity and confidence.

What You'll Learn:

- **How to set up your development environment using conda, VS Code, Git, Docker, and cloud-computing tools**
- **Core Python programming skills, from variables and data structures to file I/O and Pandas**
- **Vector and raster data processing, interactive mapping, 3D visualization, and geoprocessing**
- **Cloud-based geospatial computing with Google Earth Engine and advanced topics like hyperspectral data and spatial SQL**

Dr. Qiusheng Wu is a leading researcher and educator in open geospatial data science. He is the creator of several widely used open-source Python packages, including geemap, leafmap, segment-geospatial, and geoai. Connect with him online:

- <https://x.com/giswqs>
- <https://linkedin.com/in/giswqs>
- <https://youtube.com/@giswqs>
- <https://github.com/giswqs>



Scan the QR code to visit the
book website and download
code examples from
<https://gispro.gishub.org>