PyGIS - Open Source Spatial Programming & Remote Sensing

Contents

The globe is now digital. Everything from monitoring deforestation, predicting wildfires, to training autonomous vehicles and tracking uprisings on social media requires you to understand how to leverage location data. This book will introduce you to the methods required for spatial programming. We focus on building your core programming techniques while helping you: leverage spatial data from OSM and the US Census, use satellite imagery, track land-use change, and track social distance during a pandemic, amongst others. We will leverage open source Python packages such as GeoPandas, Rasterio, Sklearn, and Geowombat to better understand our world and help predict its future. Some Python programming experience is required, however the material will be presented in a student-friendly manner and will focus on real-world application.

Welcome - Let's get started

This section will answer a few basic questions about python:

- What is python?
- How do we use it?
- How can we extend its functionality?

Getting Started in Python

> “Python has gotten sufficiently weapons grade that we don’t descend into R anymore. Sorry, R people. I used to be one of you but we no longer descend into R.” – Chris Wiggins

What’s Python?

Python is a general-purpose programming language conceived in 1989 by Dutch programmer Guido van Rossum.

Python is free and open source, with development coordinated through the Python Software Foundation.

Python has experienced rapid adoption in the last decade and is now one of the most commonly used programming languages.

Popular textbooks on Python programming include [] and [].

Common Uses

Python is a general-purpose language used in almost all application domains such as

- communications
- web development
- CGI and graphical user interfaces
- game development
- multimedia, data processing, security, etc., etc., etc.

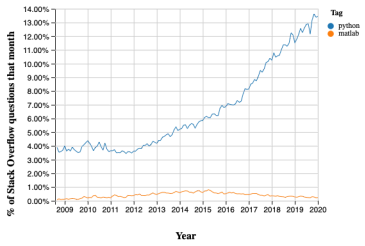
Python is beginner-friendly and routinely used to teach computer science and programming in top computer science programs.

Python is particularly popular within the scientific and data science communities.

It is steadily replacing familiar tools like Excel in the fields of finance and banking.

Relative Popularity

The following chart, produced using Stack Overflow Trends, shows one measure of the relative popularity of Python

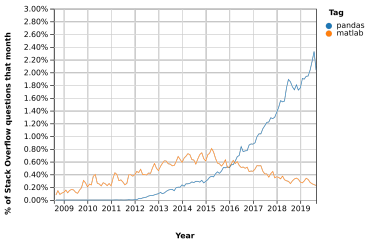


The figure indicates not only that Python is widely used but also that adoption of Python has accelerated significantly since 2012.

This is driven at least in part by uptake in the scientific domain, particularly in rapidly growing fields like data science.

For example, the popularity of pandas, a library for data analysis with Python has exploded, as seen here.

(The corresponding time path for MATLAB is shown for comparison)



Note that pandas takes off in 2012, which is the same year that we see Python’s popularity begin to spike in the first figure.

Overall, it’s clear that

- Python is one of the most popular programming languages worldwide.
- Python is a major tool for scientific computing, accounting for a rapidly rising share of scientific work around the globe.

Features

Python is a high-level language suitable for rapid development.

It has a relatively small core language supported by many libraries.

Multiple programming styles are supported (procedural, object-oriented, functional, etc.)

Python is interpreted rather than compiled.

Syntax and Design

One nice feature of Python is its elegant syntax — we’ll see many examples later on.

Elegant code might sound superfluous but in fact it’s highly beneficial because it makes the syntax easy to read and easy to remember.

Remembering how to read from files, sort dictionaries and other such routine tasks means that you don’t need to break your flow in order to hunt down correct syntax.

Closely related to elegant syntax is an elegant design.

Features like iterators, generators, decorators and list comprehensions make Python highly expressive, allowing you to get more done with less code.

Namespaces improve productivity by cutting down on bugs and syntax errors.

Scientific Programming

Python has become one of the core languages of scientific computing.

It's either the dominant player or a major player in

- machine learning and data science
- astronomy
- artificial intelligence
- chemistry
- computational biology
- meteorology

Its popularity in economics is also beginning to rise.

Bibliography

Setting up a Normal Python Environment

Overview

In this lecture, you will learn how to

1. get a Python environment up and running
2. execute simple Python commands
3. run a sample program
4. install the code libraries that underpin these lectures

Anaconda

The core Python package is easy to install but *not* what you should choose for these lectures.

These lectures require the entire scientific programming ecosystem, which

- the core installation doesn't provide
- is painful to install one piece at a time.

Hence the best approach for our purposes is to install a Python distribution that contains

1. the core Python language **and**
2. compatible versions of the most popular scientific libraries.

The best such distribution is Anaconda.

Anaconda is

- very popular
- cross-platform
- comprehensive
- completely unrelated to the Nicki Minaj song of the same name

Anaconda also comes with a great package management system to organize your code libraries.

> **Note**
>
> All of what follows assumes that you adopt this recommendation!

Installing Anaconda

To install Anaconda, download the binary and follow the instructions.

Important points:

- Install the latest version!
- If you are asked during the installation process whether you'd like to make Anaconda your default Python installation, say yes.

Updating Anaconda

Anaconda supplies a tool called `conda` to manage and upgrade your Anaconda packages.

One `conda` command you should execute regularly is the one that updates the whole Anaconda distribution.

As a practice run, please execute the following

1. Open up a terminal
2. Type `conda update anaconda`

For more information on `conda`, type `conda help` in a terminal.

Jupyter Notebooks

Jupyter notebooks are one of the many possible ways to interact with Python and the scientific libraries.

They use a *browser-based* interface to Python with

- The ability to write and execute Python commands.
- Formatted output in the browser, including tables, figures, animation, etc.

- The option to mix in formatted text and mathematical expressions.

Because of these features, Jupyter is now a major player in the scientific computing ecosystem.

Figure 1 shows the execution of some code (borrowed from here) in a Jupyter notebook

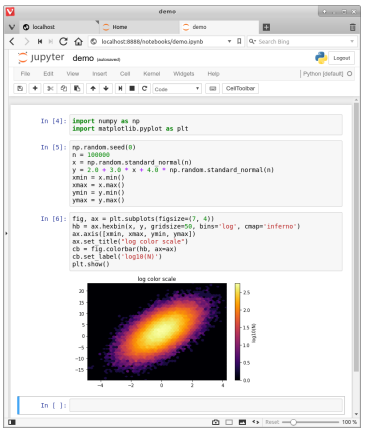


Fig. 1 A Jupyter notebook viewed in the browser

While Jupyter isn't the only way to code in Python, it's great for when you wish to

- get started
- test new ideas or interact with small pieces of code
- share scientific ideas with students or colleagues

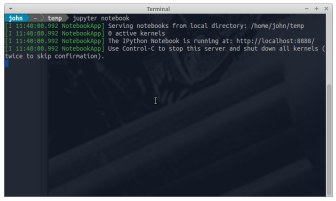
Starting the Jupyter Notebook

Once you have installed Anaconda, you can start the Jupyter notebook.

Either

- search for Jupyter in your applications menu, or
- open up a terminal and type `jupyter notebook`
    - Windows users should substitute "Anaconda command prompt" for "terminal" in the previous line.

If you use the second option, you will see something like this

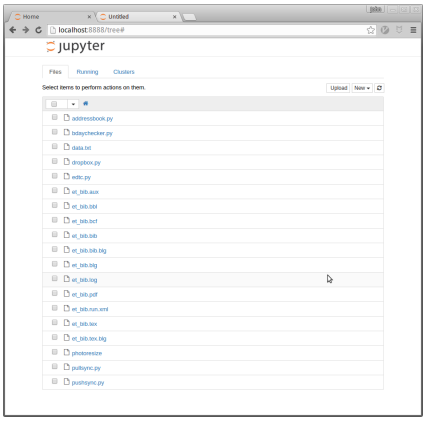


The output tells us the notebook is running at `http://localhost:8888/`

- `localhost` is the name of the local machine
- `8888` refers to port number 8888 on your computer

Thus, the Jupyter kernel is listening for Python commands on port 8888 of our local machine.

Hopefully, your default browser has also opened up with a web page that looks something like this

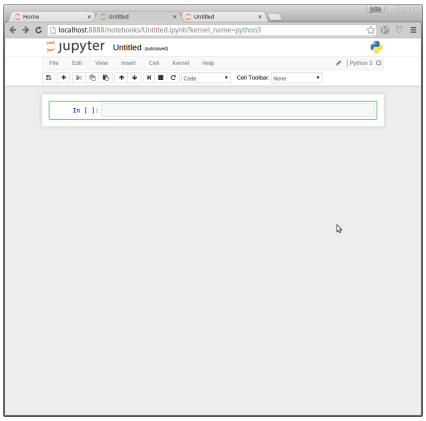


What you see here is called the Jupyter *dashboard*.

If you look at the URL at the top, it should be `localhost:8888` or similar, matching the message above.

Assuming all this has worked OK, you can now click on `New` at the top right and select `Python 3` or similar.

Here's what shows up on our machine:



The notebook displays an *active cell*, into which you can type Python commands.

Notebook Basics

Let's start with how to edit code and run simple programs.

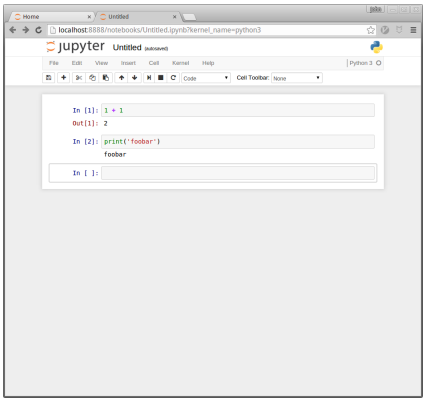
Running Cells

Notice that, in the previous figure, the cell is surrounded by a green border.

This means that the cell is in *edit mode*.

In this mode, whatever you type will appear in the cell with the flashing cursor.

When you're ready to execute the code in a cell, hit `Shift-Enter` instead of the usual `Enter`.



(Note: There are also menu and button options for running code in a cell that you can find by exploring)

Modal Editing

The next thing to understand about the Jupyter notebook is that it uses a *modal* editing system.

This means that the effect of typing at the keyboard **depends on which mode you are in**.

The two modes are

1. Edit mode
    - Indicated by a green border around one cell, plus a blinking cursor
    - Whatever you type appears as is in that cell
2. Command mode
    - The green border is replaced by a grey (or grey and blue) border
    - Keystrokes are interpreted as commands — for example, typing `b` adds a new cell below the current one

To switch to

- command mode from edit mode, hit the `Esc` key or `Ctrl-M`
- edit mode from command mode, hit `Enter` or click in a cell

The modal behavior of the Jupyter notebook is very efficient when you get used to it.

Inserting Unicode (e.g., Greek Letters)

Python supports unicode, allowing the use of characters such as α and β as names in your code.

In a code cell, try typing `\alpha` and then hitting the `tab` key on your keyboard.

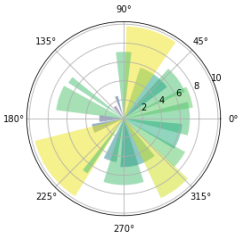
A Test Program

Let's run a test program.

Here's an arbitrary program we can use: http://matplotlib.org/3.1.1/gallery/pie_and_polar_charts/polar_bar.html.

On that page, you'll see the following code

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Fixing random state for reproducibility
np.random.seed(19680801)

# Compute pie slices
N = 20
θ = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)
colors = plt.cm.viridis(radii / 10.)

ax = plt.subplot(111, projection='polar')
ax.bar(θ, radii, width=width, bottom=0.0, color=colors, alpha=0.5)

plt.show()
```



Don't worry about the details for now — let's just run it and see what happens.

The easiest way to run this code is to copy and paste it into a cell in the notebook.

Hopefully you will get a similar plot.

Working with the Notebook

Here are a few more tips on working with Jupyter notebooks.

Tab Completion

In the previous program, we executed the line `import numpy as np`

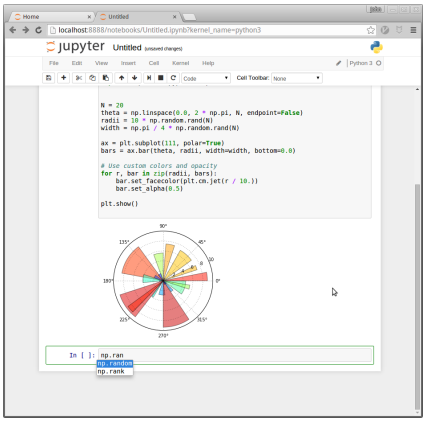
- NumPy is a numerical library we'll work with in depth.

After this import command, functions in NumPy can be accessed with `np.function_name` type syntax.

- For example, try `np.random.randn(3)`.

We can explore these attributes of `np` using the `Tab` key.

For example, here we type `np.ran` and hit Tab



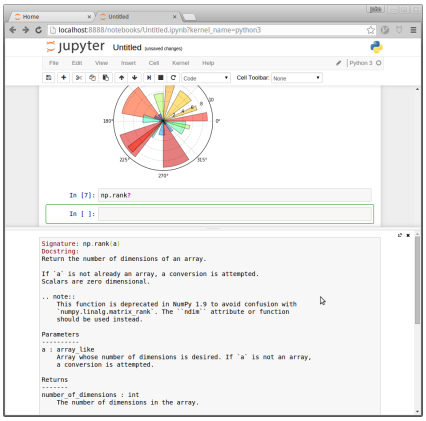
Jupyter offers up the two possible completions, `random` and `rank`.

In this way, the Tab key helps remind you of what's available and also saves you typing.

On-Line Help

To get help on `np.rank`, say, we can execute `np.rank?`.

Documentation appears in a split window of the browser, like so

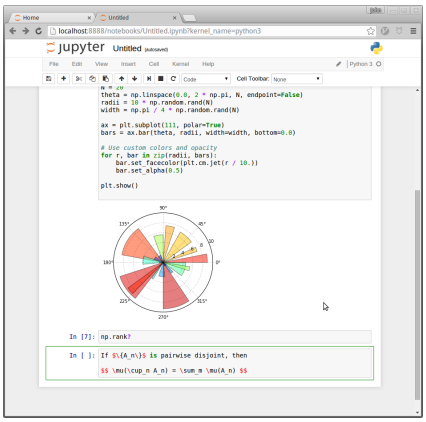


Clicking on the top right of the lower split closes the on-line help.

Other Content

In addition to executing code, the Jupyter notebook allows you to embed text, equations, figures and even videos in the page.

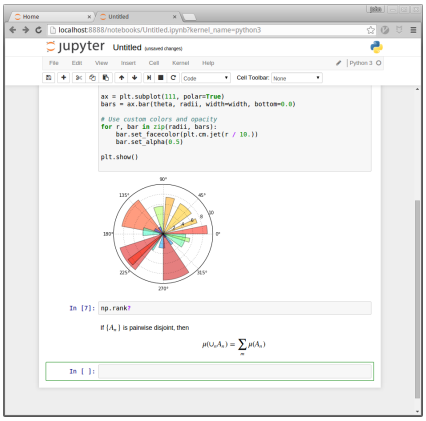
For example, here we enter a mixture of plain text and LaTeX instead of code



Next we `Esc` to enter command mode and then type `m` to indicate that we are writing Markdown, a mark-up language similar to (but simpler than) LaTeX.

(You can also use your mouse to select `Markdown` from the `Code` drop-down box just below the list of menu items)

Now we `Shift+Enter` to produce this



Sharing Notebooks

Notebook files are just text files structured in JSON and typically ending with `.ipynb`.

You can share them in the usual way that you share files — or by using web services such as nbviewer.

The notebooks you see on that site are **static** html representations.

To run one, download it as an `ipynb` file by clicking on the download icon.

Save it somewhere, navigate to it from the Jupyter dashboard and then run as discussed above.

> **Learning Objectives**
>
> - Learn the basics of Docker
> - Pull, run, and update a container set up for spatial modeling

Review

- A normal intro to python environments

Setting up Python for Spatial on Mac, Windows, and Linux

Spatial analysis requires a pretty broad set of python modules and with it, comes a lot of dependencies. And to be honest, the only thing Python doesn't do well with, is dependencies. Luckily we have a few tricks up our sleeves to help you get to work fast.

Docker for Spatial Python - GDAL Included

Docker allows us to essentially package and share operating systems with specific modifications. Importantly for us this includes libraries and dependencies that are difficult to install otherwise (I'm looking at you GDAL). Before we start you should familiarize yourself with the basic concepts behind Docker, please read the following: a simple intro to Docker concepts

To get this done we will be accessing DockerHub, which allows coders like us to store their Docker images. We will be downloading an image of the Linux operating system (Ubuntu, which is "debian-based"), that already has GDAL built for us. Once we "pull" a copy of the image of this operating system we will open it as "a container". This "container" is a running instance of the "image" that we can run our applications on, and customize for our use case. Read some more on "images" vs "containers" etc here.

Install Docker

Follow the instructions for installing Docker on your system, take your time, and make sure you understand what you are doing before you do it!

Mac | Windows | Linux

```
https://docs.docker.com/desktop/mac/install/
```

There are two ways we can do this, the easy way, or the much more detailed way. Choose your poison.

The Easy Way

Pulling a Docker Image Ready for GIS

Now we are going to get an Ubuntu image running that already has everything installed and ready to use. We are going to pull the lastest build of our image from my docker hub page.

First open a **terminal** or **powershell** in windows… yes powershell not terminal, type the following:

Mac | Windows | Linux

```
# download the image
docker pull mmann1123/gw_pygis

# list your images
docker images -a

# you should see mmann1123/gw_pygis
```

Now we can access python through jupyter notebooks (read about jupyter notebooks here). Jupyter is probably the easiest way to start your coding.

Note

You can mount a volume from your normal operating system to your linux container using the `-v` option of `docker run`. In the above case you can connect your `/Users/<user_name>/Documents` folder into the `/home` folder of your container by running `docker run -v /Users/<user_name>/Documents:/home` (mac), `docker run -v //c/User/<user>/Documents:/home` (windows), or `docker run -v /home/<user_name>/Documents:/home` (linux). To access your documents folder from within you container just `cd` into it e.g. `cd /home`.

To do this we are going to attach a local volume with `-v`, open a port with `-p` and run mmann1123/gw_pygis, once inside of the runnning linux computer we will launch jupyter notebook and set an ip address to access it using `-ip`, and we will allow administrative privileges using `--allow-root`. After executing the code we simply need to open up the URL displayed in response.

Mac | Windows | Linux

```
# Or if browser is  present
docker run -v /Users/<user_name>/path_to_folder_you_want_access_to:/home  -it -p 8888:8888 mmann1123/gw_pygis
jupyter notebook --ip 0.0.0.0  --allow-root

# or if the jupyter notebooks doesn't launch automatically
docker run -v /Users/<user_name>/path_to_folder_you_want_access_to:/home  -it -p 8888:8888 mmann1123/gw_pygis
jupyter notebook --ip 0.0.0.0 --no-browser --allow-root
# THEN control click on URL printed to the bottom of terminal
```

Every time you want to run pygis you are going to `run` the docker container called `mmann1123/gw_pygis`.

Warning

When working in your container make sure to store all your data outside of the container! This is kind of like a school computer, where every time you log out, all the changes you made are deleted.

You can save your data in your linked volume which in these examples can be found by typing `cd /home/` while inside your container. The connected folder was defined with the `-v /home/<user_name>/path_to_folder_you_want_access_to:/home` option with docker run.

To make this a little easier you can create an executable script on your desktop to run it when you want.

Mac Bash | Windows | Linux

```
# move to your desktop
```

```
cd ~/Desktop/

# write a shell script called run_pygis
# between the ''s put whatever bash code you want
echo '
sudo docker run -v /Users/<user_name>/path_to_folder_you_want_access_to:/home  -it -p 8888:8888 mmann1123/gw_pygis
jupyter notebook --ip 0.0.0.0  --allow-root
' > run_pygis.sh

# allow it to be executable
chmod 755 run_pygis.sh
```

Now that we have an executable file we need to execute this, on linux at least, the only one I can test on, we need to execute it from the command line. But its pretty easy.

To execute our run_geowombat.sh script we need to navigate to the Desktop in your local terminal, then execute the file:

Mac Bash **Windows** **Linux**

```
# move to your desktop
cd ~/Desktop/

# execute it
./run_pygis.sh
```

When you're done with your work inside the container, and double checked that your data is saved locally - not on the container - you can type `exit` in the terminal window to exit your geowombat container.

The More Detailed Way

Pull and Run a Linux Image with GDAL

Now we are going to get an Ubuntu image running that already has GDAL installed. We are going to pull the lastest build of our image from [OSGEO's docker hub page](#).

First open a terminal or powershell, type the following:

Mac **Windows** **Linux**

```
# download the image
docker pull osgeo/gdal:ubuntu-full-latest

# list your images
docker images -a

# run osgeo/gdal image, but link my volume /your_folder_to_share_with_image:/location_on_container_to_access_it
# here I am linking my <user_name> home folder to the containers home folder
# important: update the <user_name> portion with your windows user name
docker run -v /Users/<user_name>/path_to_folder_you_want_access_to:/home  -it osgeo/gdal:ubuntu-full-latest
```

> **Note**
>
> You can mount a volume from your normal operating system to your linux container using the `-v` option of `docker run`. In the above case you can connect your `C:/User/<user>/Documents` folder into the `/home` folder of your container by running `docker run -v //c/User/<user>/Documents:/home`. To access your documents folder from within you container just `cd` into it e.g. `cd /home`.

Your command prompt in the terminal window should now say something funny like `root@b0c5ab799195:/#` . You are now INSIDE your running docker container, which is running Ubuntu linux.

Just to demonstrate this is really linux, let's ask the system what OS is running:

Linux Container

```
uname
```

Should return `Linux`

We will need to update/upgrade the OS, install pip and a few other applications we need, pip install geowombat, and a few more python dependancies for it. We will then exit the container and save a named copy of it.

Linux Container

```
# update Ubuntu
apt update -y
apt upgrade -y

# install pip and a few other things
apt install python3-pip git gdal-bin libgdal-dev libspatialindex-dev   -y

# install geowombat and with it geopandas, rasterio etc
pip install git+https://github.com/jgrss/geowombat

# install a few more dependancies for geowombat, including jupyter notebooks
pip install cython numpy retry requests opencv-python notebook

#test and exit - this should print the version of geowombat installed
python -c "import geowombat as gw;print(gw.__version__)"

# Install any other modules you want via pip
```

Now we need to exit the container and go back to your local computer command line.

Linux Container

```
exit
```

Note that we are now in your boring old terminal/shell, we exited Ubuntu.

We now need to create a named image that includes geowombat etc. Without doing this, the next time we start the osgeo/gdal image nothing will have been saved.

Mac Windows Linux

```
# list all available containers
docker ps -a

# find the "CONTAINER ID" of the container that was just exited seconds ago,
# and replace the example ID used below

# commit changes to new named image (replace 12 digit container id from one listed above)
docker commit 9c3f33afcff9 pygis

# list all available images, look for pygis.
docker images
```

Ok, now we are getting somewhere. We have created a new image called “pygis” that should have everything we need to get this done! Now the problem is how to access it?!

> **Note**
>
> Keep in mind if you want to make changes to the ‘pygis’ image you will need to first run it via the command line, make your changes, and then ‘commit’ or save another named version (ideally with a different name)

Access your spatial python image

There are two ways to access this 1) through the command-line and 2) through Jupyter Notebooks.

Let’s start with command line only access. Note that this is almost exactly how we ran the osgeo/gdal image, except we replaced its name with geowombat. *Don’t forget to replace `<user_name>` with your user name!*

Mac Windows Linux

```
docker run -v /Users/<user_name>/path_to_folder_you_want_access_to:/home  -it pygis
python
```

Now you are at the python command prompt, start coding!

Alternatively, we can access python through jupyter notebooks (read about jupyter notebooks here). Jupyter is probably the easiest way to start your coding.

To do this we are going to

Mac Windows Linux

```
# Or if browser is  present
docker run -v /Users/<user_name>/path_to_folder_you_want_access_to:/home  -it -p 8888:8888 pygis
jupyter notebook --ip 0.0.0.0  --allow-root

# or if the jupyter notebooks doesn't launch automatically
docker run -v /Users/<user_name>/path_to_folder_you_want_access_to:/home  -it -p 8888:8888 pygis
jupyter notebook --ip 0.0.0.0 --no-browser --allow-root
# THEN control click on URL printed to the bottom of terminal
```

Every time you want to run pygis you are going to `run` the docker container called `pygis`.

> **Warning**
>
> **When working in your container make sure to store all your data outside of the container!** This is kind of like a school computer, where every time you log out, all the changes you made are deleted.
>
> You can save your data in your linked volume which in these examples can be found by typing `cd /home/` while inside your container. The connected folder was defined with the `-v /home/<user_name>/path_to_folder_you_want_access_to:/home` option with docker run.

To make this a little easier you can create an executable script on your desktop to run it when you want.

Mac Bash Windows Linux

```
# move to your desktop
cd ~/Desktop/

# write a shell script called run_pygis
# between the ''s put whatever bash code you want
echo '
docker run -v /Users/<user_name>/path_to_folder_you_want_access_to:/home  -it -p 8888:8888 pygis
jupyter notebook --ip 0.0.0.0  --allow-root
' > run_pygis.sh

# allow it to be executable
chmod 755 run_pygis.sh
```

Now that we have an executable file we need to execute this, on linux at least, the only one I can test on, we need to execute it from the command line. But its pretty easy.

To execute our run_geowombat.sh script we need to navigate to the Desktop, then execute the file:

Mac Bash Windows Linux

```
# move to your desktop
cd ~/Desktop/

# execute it
./run_pygis.sh
```

When you’re done with your work inside the container, and double checked that your data is saved locally - not on the container - you can type `exit` in the terminal window to exit your geowombat container.

An Introductory Example

Overview

We're now ready to start learning the Python language itself.

In this lecture, we will write and then pick apart small Python programs.

The objective is to introduce you to basic Python syntax and data structures.

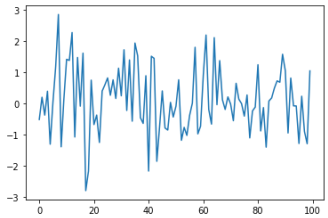
Deeper concepts will be covered in later lectures.

You should have read the lecture on getting started with Python before beginning this one.

The Task: Plotting a White Noise Process

Suppose we want to simulate and plot the white noise process $\epsilon_0, \epsilon_1, \ldots, \epsilon_T$, where each draw ϵ_t is independent standard normal.

In other words, we want to generate figures that look something like this:



(Here t is on the horizontal axis and ϵ_t is on the vertical axis.)

We'll do this in several different ways, each time learning something more about Python.

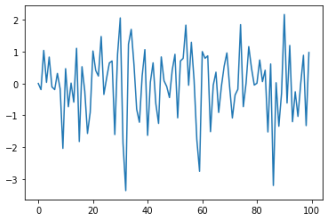
We run the following command first, which helps ensure that plots appear in the notebook if you run it on your own machine.

```
%matplotlib inline
```

Version 1

Here are a few lines of code that perform the task we set

```
import numpy as np
import matplotlib.pyplot as plt

ϵ_values = np.random.randn(100)
plt.plot(ϵ_values)
plt.show()
```



Let's break this program down and see how it works.

Imports

The first two lines of the program import functionality from external code libraries.

The first line imports NumPy, a favorite Python package for tasks like

- working with arrays (vectors and matrices)
- common mathematical functions like `cos` and `sqrt`
- generating random numbers
- linear algebra, etc.

After `import numpy as np` we have access to these attributes via the syntax `np.attribute`.

Here's two more examples

```
np.sqrt(4)
```

```
2.0
```

```
np.log(4)
```

```
1.3862943611198906
```

We could also use the following syntax:

```
import numpy

numpy.sqrt(4)
```

```
2.0
```

But the former method (using the short name `np`) is convenient and more standard.

Why So Many Imports?

Python programs typically require several import statements.

The reason is that the core language is deliberately kept small, so that it's easy to learn and maintain.

When you want to do something interesting with Python, you almost always need to import additional functionality.

Packages

As stated above, NumPy is a Python *package*.

Packages are used by developers to organize code they wish to share.

In fact, a package is just a directory containing

1. files with Python code — called **modules** in Python speak
2. possibly some compiled code that can be accessed by Python (e.g., functions compiled from C or FORTRAN code)
3. a file called `__init__.py` that specifies what will be executed when we type `import package_name`

In fact, you can find and explore the directory for NumPy on your computer easily enough if you look around.

On this machine, it's located in

```
anaconda3/lib/python3.7/site-packages/numpy
```

Subpackages

Consider the line `ϵ_values = np.random.randn(100)`.

Here `np` refers to the package NumPy, while `random` is a **subpackage** of NumPy.

Subpackages are just packages that are subdirectories of another package.

Importing Names Directly

Recall this code that we saw above

```
import numpy as np

np.sqrt(4)
```

```
2.0
```

Here's another way to access NumPy's square root function

```
from numpy import sqrt

sqrt(4)
```

```
2.0
```

This is also fine.

The advantage is less typing if we use `sqrt` often in our code.

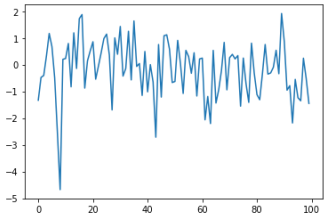
The disadvantage is that, in a long program, these two lines might be separated by many other lines.

Then it's harder for readers to know where `sqrt` came from, should they wish to.

Random Draws

Returning to our program that plots white noise, the remaining three lines after the import statements are

```
ϵ_values = np.random.randn(100)
plt.plot(ϵ_values)
plt.show()
```



The first line generates 100 (quasi) independent standard normals and stores them in `ϵ_values`.

The next two lines genererate the plot.

We can and will look at various ways to configure and improve this plot below.

Alternative Implementations

Let's try writing some alternative versions of our first program, which plotted IID draws from the normal distribution.

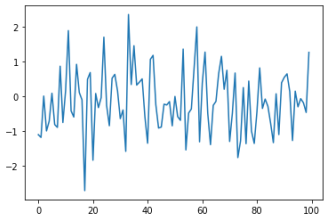
The programs below are less efficient than the original one, and hence somewhat artificial.

But they do help us illustrate some important Python syntax and semantics in a familiar setting.

A Version with a For Loop

Here's a version that illustrates `for` loops and Python lists.

```
ts_length = 100
ϵ_values = []   # empty list

for i in range(ts_length):
    e = np.random.randn()
    ϵ_values.append(e)

plt.plot(ϵ_values)
plt.show()
```



In brief,

- The first line sets the desired length of the time series.
- The next line creates an empty *list* called `ϵ_values` that will store the ϵ_t values as we generate them.
- The statement `# empty list` is a *comment*, and is ignored by Python's interpreter.
- The next three lines are the `for` loop, which repeatedly draws a new random number ϵ_t and appends it to the end of the list `ϵ_values`.
- The last two lines generate the plot and display it to the user.

Let's study some parts of this program in more detail.

Lists

Consider the statement `ϵ_values = []`, which creates an empty list.

Lists are a *native Python data structure* used to group a collection of objects.

For example, try

```
x = [10, 'foo', False]
type(x)
```

```
list
```

The first element of `x` is an integer, the next is a string, and the third is a Boolean value.

When adding a value to a list, we can use the syntax `list_name.append(some_value)`

```
x
```

```
[10, 'foo', False]
```

```
x.append(2.5)
x
```

```
[10, 'foo', False, 2.5]
```

Here `append()` is what's called a *method*, which is a function "attached to" an object—in this case, the list `x`.

We'll learn all about methods later on, but just to give you some idea,

- Python objects such as lists, strings, etc. all have methods that are used to manipulate the data contained in the object.
- String objects have string methods, list objects have list methods, etc.

Another useful list method is `pop()`

```
x
```

```
[10, 'foo', False, 2.5]
```

```
x.pop()
```

```
2.5
```

```
x
```

```
[10, 'foo', False]
```

Lists in Python are zero-based (as in C, Java or Go), so the first element is referenced by `x[0]`

```
x[0]   # first element of x
```

```
10
```

```
x[1]   # second element of x
```

```
'foo'
```

The For Loop

Now let's consider the `for` loop from the program above, which was

```
for i in range(ts_length):
    e = np.random.randn()
    ϵ_values.append(e)
```

Python executes the two indented lines `ts_length` times before moving on.

These two lines are called a `code block`, since they comprise the “block” of code that we are looping over.

Unlike most other languages, Python knows the extent of the code block *only from indentation*.

In our program, indentation decreases after line `ϵ_values.append(e)`, telling Python that this line marks the lower limit of the code block.

More on indentation below—for now, let’s look at another example of a `for` loop

```
animals = ['dog', 'cat', 'bird']
for animal in animals:
    print("The plural of " + animal + " is " + animal + "s")
```

```
The plural of dog is dogs
The plural of cat is cats
The plural of bird is birds
```

This example helps to clarify how the `for` loop works: When we execute a loop of the form

```
for variable_name in sequence:
    <code block>
```

The Python interpreter performs the following:

- For each element of the `sequence`, it “binds” the name `variable_name` to that element and then executes the code block.

The `sequence` object can in fact be a very general object, as we’ll see soon enough.

A Comment on Indentation

In discussing the `for` loop, we explained that the code blocks being looped over are delimited by indentation.

In fact, in Python, **all** code blocks (i.e., those occurring inside loops, if clauses, function definitions, etc.) are delimited by indentation.

Thus, unlike most other languages, whitespace in Python code affects the output of the program.

Once you get used to it, this is a good thing: It

- forces clean, consistent indentation, improving readability
- removes clutter, such as the brackets or end statements used in other languages

On the other hand, it takes a bit of care to get right, so please remember:

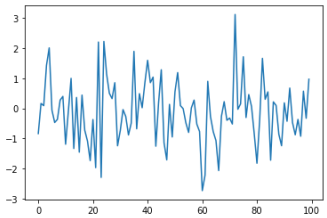
- The line before the start of a code block always ends in a colon
  - `for i in range(10):`
  - `if x > y:`
  - `while x < 100:`
  - etc., etc.
- All lines in a code block **must have the same amount of indentation**.
- The Python standard is 4 spaces, and that’s what you should use.

While Loops

The `for` loop is the most common technique for iteration in Python.

But, for the purpose of illustration, let’s modify the program above to use a `while` loop instead.

```
ts_length = 100
ϵ_values = []
i = 0
while i < ts_length:
    e = np.random.randn()
    ϵ_values.append(e)
    i = i + 1
plt.plot(ϵ_values)
plt.show()
```



Note that

- the code block for the `while` loop is again delimited only by indentation
- the statement `i = i + 1` can be replaced by `i += 1`

Another Application

Let’s do one more application before we turn to exercises.

In this application, we plot the balance of a bank account over time.

There are no withdraws over the time period, the last date of which is denoted by T.

The initial balance is b_0 and the interest rate is r.

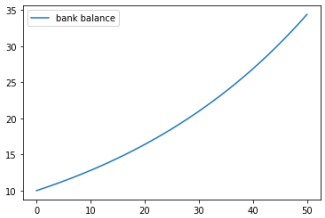
The balance updates from period t to $t+1$ according to

$$b_{t+1} = (1 + r)b_t \tag{1}$$

In the code below, we generate and plot the sequence $b_0, b_1, \ldots, b_T$ generated by (1).

Instead of using a Python list to store this sequence, we will use a NumPy array.

```
r = 0.025         # interest rate
T = 50            # end date
b = np.empty(T+1) # an empty NumPy array, to store all b_t
b[0] = 10         # initial balance

for t in range(T):
    b[t+1] = (1 + r) * b[t]

plt.plot(b, label='bank balance')
plt.legend()
plt.show()
```



The statement `b = np.empty(T+1)` allocates storage in memory for `T+1` (floating point) numbers.

These numbers are filled in by the `for` loop.

Allocating memory at the start is more efficient than using a Python list and `append`, since the latter must repeatedly ask for storage space from the operating system.

Notice that we added a legend to the plot.

Learn More

We're about ready to wrap up this brief course on Python for scientific computing.

In this last lecture we give some pointers to the major scientific libraries and suggestions for further reading.

NumPy

Fundamental matrix and array processing capabilities are provided by the excellent NumPy library.

For example, let's build some arrays

```
import numpy as np                    # Load the library

a = np.linspace(-np.pi, np.pi, 100)   # Create even grid from -π to π
b = np.cos(a)                         # Apply cosine to each element of a
c = np.sin(a)                         # Apply sin to each element of a
```

Now let's take the inner product

```
b @ c
```

```
4.04891256782214e-16
```

The number you see here might vary slightly due to floating point arithmetic but it's essentially zero.

As with other standard NumPy operations, this inner product calls into highly optimized machine code.

It is as efficient as carefully hand-coded FORTRAN or C.

SciPy

The SciPy library is built on top of NumPy and provides additional functionality.

For example, let's calculate $\int_{-2}^{2} \phi(z)dz$ where ϕ is the standard normal density.

```
from scipy.stats import norm
from scipy.integrate import quad

ϕ = norm()
value, error = quad(ϕ.pdf, -2, 2)  # Integrate using Gaussian quadrature
value
```

```
0.9544997361036417
```

SciPy includes many of the standard routines used in

- linear algebra
- integration
- interpolation
- optimization
- distributions and random number generation
- signal processing

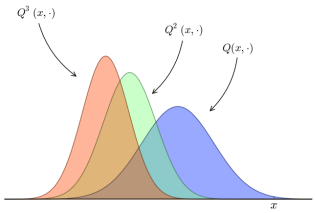
See them all here.

Graphics

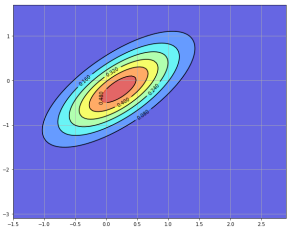
The most popular and comprehensive Python library for creating figures and graphs is Matplotlib, with functionality including

- plots, histograms, contour images, 3D graphs, bar charts etc.
- output in many formats (PDF, PNG, EPS, etc.)
- LaTeX integration

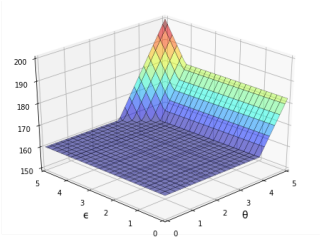
Example 2D plot with embedded LaTeX annotations



Example contour plot



Example 3D plot



More examples can be found in the Matplotlib thumbnail gallery.

Other graphics libraries include

- Plotly
- Bokeh

Symbolic Algebra

It's useful to be able to manipulate symbolic expressions, as in Mathematica or Maple.

The SymPy library provides this functionality from within the Python shell.

```
from sympy import Symbol

x, y = Symbol('x'), Symbol('y')  # Treat 'x' and 'y' as algebraic symbols
x + x + x + y
```

$$3x + y$$

We can manipulate expressions

```
expression = (x + y)**2
expression.expand()
```

$$x^2 + 2xy + y^2$$

solve polynomials

```
from sympy import solve

solve(x**2 + x + 2)
```

```
[-1/2 - sqrt(7)*I/2, -1/2 + sqrt(7)*I/2]
```

and calculate limits, derivatives and integrals

```
from sympy import limit, sin, diff

limit(1 / x, x, 0)
```

$$\infty$$

```
limit(sin(x) / x, x, 0)
```

$$1$$

```
diff(sin(x), x)
```

$$\cos\left(x\right)$$

The beauty of importing this functionality into Python is that we are working within a fully fledged programming language.

We can easily create tables of derivatives, generate LaTeX output, add that output to figures and so on.

Pandas

One of the most popular libraries for working with data is pandas.

Pandas is fast, efficient, flexible and well designed.

Here's a simple example, using some dummy data generated with Numpy's excellent `random` functionality.

```
import pandas as pd
np.random.seed(1234)

data = np.random.randn(5, 2)  # 5x2 matrix of N(0, 1) random draws
dates = pd.date_range('28/12/2010', periods=5)

df = pd.DataFrame(data, columns=('price', 'weight'), index=dates)
print(df)
```

```
               price    weight
2010-12-28  0.471435 -1.190976
2010-12-29  1.432707 -0.312652
2010-12-30 -0.720589  0.887163
2010-12-31  0.859588 -0.636524
2011-01-01  0.015696 -2.242685
```

```
df.mean()
```

```
price     0.411768
weight   -0.699135
dtype: float64
```

Further Reading

These lectures were originally taken from a longer and more complete lecture series on Python programming hosted by QuantEcon.

The full set of lectures might be useful as the next step of your study.

Learning Objectives

- Learn the difference between raster and vector data
- Learn how to manually create each data structure
- Learn about data measurement

Spatial Data

Vector vs. Raster Data

To work in a GIS environment, real world observations (objects or events that can be recorded in 2D or 3D space) need to be reduced to spatial entities. These spatial entities can be represented in a GIS as a **vector data model** or a **raster data model**.

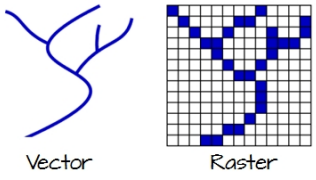


Fig. 2 Vector and raster representations of a river feature.

Vector Data

Vector features can be decomposed into three different geometric primitives: **points**, **polylines** and **polygons**.

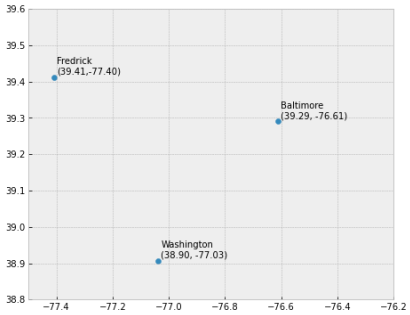
Point

```
import geopandas as gpd
import matplotlib.pyplot as plt
from shapely.geometry import Point

d = {'name': ['Washington\n(38.90, -77.03)', 'Baltimore\n(39.29,
-76.61)','Fredrick\n(39.41,-77.40)'],
     'geometry': [Point(-77.036873,38.907192), Point(-76.612190,39.290386,),
Point(-77.408456,39.412006)]}

gdf = gpd.GeoDataFrame(d, crs="EPSG:4326")
print(gdf)
```

```
                        name                    geometry
0  Washington\n(38.90, -77.03)  POINT (-77.03687 38.90719)
1   Baltimore\n(39.29, -76.61)  POINT (-76.61219 39.29039)
2     Fredrick\n(39.41,-77.40)  POINT (-77.40846 39.41201)
```

A point is composed of one coordinate pair representing a specific location in a coordinate system. Points are the most basic geometric primitives having no length or area. By definition a point can't be "seen" since it has no area; but this is not practical if such primitives are to be mapped. So points on a map are represented using *symbols* that have both area and shape (e.g. circle, square, plus signs).

```
plt.style.use('bmh') # better for plotting geometries vs general plots.

fig, ax = plt.subplots(figsize=(12, 6))
gdf.plot(ax=ax)
plt.ylim([38.8, 39.6])
plt.xlim([-77.5, -76.2])

for x, y, label in zip(gdf.geometry.x, gdf.geometry.y, gdf.name):
    ax.annotate(label, xy=(x, y), xytext=(3, 3), textcoords="offset points")
plt.show()
```

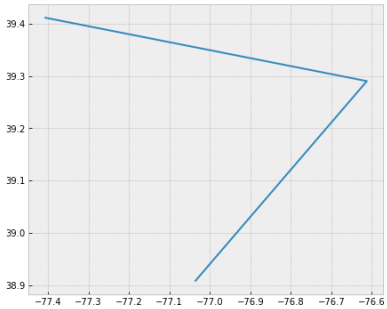


We seem capable of interpreting such symbols as points, but there may be instances when such interpretation may be ambiguous (e.g. is a round symbol delineating the area of a round feature on the ground such as a large oil storage tank or is it representing the point location of that tank?).

Polyline

```
from shapely.geometry import LineString

d = {'name': ['Washington\n(38.90, -77.03)' ],
     'geometry': [LineString([Point(-77.036873,38.907192),
Point(-76.612190,39.290386,), Point(-77.408456,39.412006)])]}
gdf = gpd.GeoDataFrame(d, crs="EPSG:4326")
fig, ax = plt.subplots(figsize=(12, 6))
gdf.plot(ax=ax)
```

```
<AxesSubplot:>
```

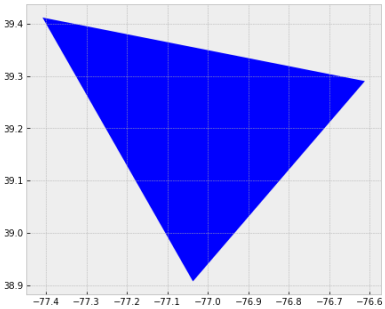


A polyline is composed of a sequence of two or more coordinate pairs called vertices. A vertex is defined by coordinate pairs, just like a point, but what differentiates a vertex from a point is its explicitly defined relationship with neighboring vertices. A vertex is connected to at least one other vertex.

Like a point, a true line can't be seen since it has no area. And like a point, a line is symbolized using shapes that have a color, width and style (e.g. solid, dashed, dotted, etc…). Roads and rivers are commonly stored as polylines in a GIS.

Polygon

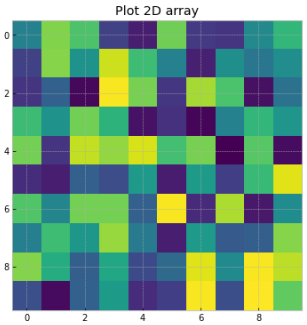
```
from shapely.geometry import Polygon

d = {'name': ['Washington\n(38.90, -77.03)' ],
     'geometry': [Polygon([Point(-77.036873,38.907192),
Point(-76.612190,39.290386,), Point(-77.408456,39.412006)])]}
gdf = gpd.GeoDataFrame(d, crs="EPSG:4326")
fig, ax = plt.subplots(figsize=(12, 6))
gdf.plot(ax=ax)
plt.show()
```



A polygon is composed of three or more line segments whose starting and ending coordinate pairs are the same. Sometimes you will see the words *lattice* or *area* used in lieu of 'polygon'. Polygons represent both length (i.e. the perimeter of the area) and area. They also embody the idea of an inside and an outside; in fact, the area that a polygon encloses is explicitly defined in a GIS environment. If it isn't, then you are working with a polyline feature. If this does not seem intuitive, think of three connected lines defining a triangle: they can represent three connected road segments (thus polyline features), or they can represent the grassy strip enclosed by the connected roads (in which case an 'inside' is implied thus defining a polygon).

Raster Data

```
import numpy as np

X=np.random.randint(256, size=(10, 10))

fig = plt.figure(figsize=(8,6))
plt.imshow(X)
plt.title("Plot 2D array")
plt.show()
```



A raster data model uses an array of cells, or pixels, to represent real-world objects. Raster datasets are commonly used for representing and managing imagery, surface temperatures, digital elevation models, and numerous other entities.

A raster can be thought of as a special case of an area object where the area is divided into a regular grid of cells. But a regularly spaced array of marked points may be a better analogy since rasters are stored as an array of values where each cell is defined by a single coordinate pair inside of most GIS environments.

Implicit in a raster data model is a value associated with each cell or pixel. This is in contrast to a vector model that may or may not have a value associated with the geometric primitive.

Object vs. Field

The traditional vector/raster perspective of our world is one that has been driven by software and data storage environments. But this perspective is not particularly helpful if one is interested in analyzing the pattern. In fact, it can mask some important properties of the entity being studied. An object vs. field view of the world proves to be more insightful even though it may seem more abstract.

Object View

An object view of the world treats entities as discrete objects; they need not occur at every location within a study area. Point locations of cities would be an example of an object. So would be polygonal representations of urban areas which may be non-contiguous.

Field View

A field view of the world treats entities as a scalar field. This is a mathematical concept in which a scalar is a quantity having a magnitude. It is measurable at every location within the study region. Two popular examples of a scalar field are surface elevation and surface temperature. Each represents a property that can be measured at any location.

Another example of a scalar field is the presence and absence of a building. This is a binary scalar where a value of 0 is assigned to a location devoid of buildings and a value of 1 is assigned to locations having one or more buildings. A field representation of buildings may not seem intuitive, in fact, given the definition of an object view of the world in the last section, it would seem only fitting to view buildings as objects. In fact, buildings can be viewed as both field or objects. The context of the analysis is ultimately what will dictate which view to adopt. If we're interested in studying the distribution of buildings over a study area, then an object view of the features makes sense. If, on the other hand, we are interested in identifying all locations where buildings don't exist, then a binary field view of these entities would make sense.

Scale

How one chooses to represent a real-world entity will be in large part dictated by the **scale** of the analysis. In a GIS, scale has a specific meaning: it's the ratio of distance on the map to that in the real world. So a **large scale** map implies a relatively large ratio and thus a small extent. This is counter to the layperson's interpretation of *large scale* which focuses on the scope or extent of a study; so a large scale analysis would imply one that covers a *large* area.

The following two maps represent the same entity: the Boston region. At a small scale (e.g. 1:10,000,000), Boston and other cities may be best represented as points. At a large scale (e.g. 1:34,000), Boston may be best represented as a polygon. Note that at this large scale, roads may also be represented as polygon features instead of polylines.

Fig. 3 Map of the Boston area at a 1:10,000,000 scale. Note that in geography, this is considered small scale whereas in layperson terms, this extent is often referred to as a large scale (i.e. covering a large area).

Fig. 4 Map of the Boston area at a 1:34,000 scale. Note that in geography, this is considered large scale whereas in layperson terms, this extent is often referred to as a small scale (i.e. covering a small area).

Attribute Tables

Non-spatial information associated with a spatial feature is referred to as an **attribute**. A feature on a GIS map is linked to its record in the attribute table by a unique numerical feature identifier (FID). Every feature in a layer has an identifier. It is important to understand the one-to-one or many-to-one relationship between feature, and attribute record. Because features on the map are linked to their records in the table, many GIS software will allow you to click on a map feature and see its related attributes in the table.

Raster data can also have attributes only if pixels are represented using a small set of unique integer values. Raster datasets that contain attribute tables typically have cell values that represent or define a class, group, category, or membership. NOTE: not all GIS raster data formats can store attribute information; in fact most raster datasets you will work with in this course will not have attribute tables.

Measurement Levels

Attribute data can be broken down into four **measurement levels**:

- **Nominal** data which have no implied order, size or quantitative information (e.g. paved and unpaved roads)
- **Ordinal** data have an implied order (e.g. ranked scores), however, we cannot quantify the difference since a linear scale is not implied.
- **Interval** data are numeric and have a linear scale, however they do not have a true zero and can therefore not be used to measure *relative* magnitudes. For example, one cannot say that 60°F is twice as warm as 30°F since when presented in degrees °C the temperature values are 15.5°C and -1.1°C respectively (and 15.5 is clearly not twice as big as -1.1).
- **Ratio** scale data are interval data with a true zero such as monetary value (e.g. 1,20, $100).

Data type

Another way to categorize an attribute is by its **data type**. ArcGIS supports several data types such as **integer, float, double and text**. Knowing your data type and measurement level should dictate how they are stored in a GIS environment. The following table lists popular data types available in most GIS applications.

Type	Stored values	Note
Short integer	-32,768 to 32,768	Whole numbers
Long integer	-2,147,483,648 to 2,147,483,648	Whole numbers
Float	-3.4 * E-38 to 1.2 E38	Real numbers
Double	-2.2 * E-308 to 1.8 * E308	Real numbers
Text	Up to 64,000 characters	Letters and words

While whole numbers can be stored as a float or double (i.e. we can store the number 2 as 2.0) doing so comes at a cost: an increase in storage space. This may not be a big deal if the dataset is small, but if it consists of tens of thousands of records the increase in file size and processing time may become an issue.

While storing an integer value as a float may not have dire consequences, the same cannot be said of storing a float as an integer. For example, if your values consist of 0.2, 0.01, 0.34, 0.1 and 0.876, their integer counterpart would be 0, 0, 0, and 1 (i.e. values rounded to the nearest whole number).

Learning Objectives

- Learn about spatial data storage formats

Data Storage Formats

Vector Data File Formats

GeoJSON

GeoJSON is an open standard format designed for representing simple geographical features, along with their non-spatial attributes. It is based on the JSON format. The features include points, line strings, polygons, and multi-part collections of these types. One of it's primary advantages is that it is human readible and stores all the relevant data in a single text file. As result however, these files can get very large when storing complex geometries.

Here's a simple example of a FeatureCollection that includes a point, line and polygon `geometry` with some attributes stored as `properties`:

```
{ "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "geometry": {"type": "Point", "coordinates": [102.0, 0.5]},
      "properties": {"prop0": "value0"}
      },
    { "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]
          ]
        },
      "properties": {
        "prop0": "value0",
        "prop1": 0.0
        }
      },
    { "type": "Feature",
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],
            [100.0, 1.0], [100.0, 0.0] ]
          ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": {"this": "that"}
        }
      }
    ]
  }
```

GeoPackage

This is a relatively new data format that follows open format standards (i.e. it is non-proprietary). It's built on top of SQLite (a self-contained relational database). Its one big advantage over many other vector formats is its compactness–coordinate value, metadata, attribute table, projection information, etc…, are all stored in a *single* file which facilitates portability. Its filename usually ends in `.gpkg`. Applications such as QGIS (2.12 and up), R and ArcGIS will recognize this format (ArcGIS version 10.2.2 and above will read the file from ArcCatalog but requires a script to create a GeoPackage).

Shapefile

A **shapefile** is a file-based data format native to ArcView 3.x software (a much older version of ArcMap). Conceptually, a shapefile is a feature class–it stores a collection of features that have the same geometry type (point, line, or polygon), the same attributes, and a common spatial extent.

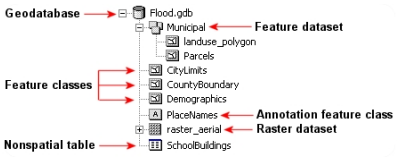
Despite what its name may imply, a "single" shapefile is actually composed of at least three files, and as many as eight. Each file that makes up a "shapefile" has a common filename but different extension type.

The list of files that define a "shapefile" are shown in the following table. Note that each file has a specific role in defining a shapefile.

File extension	Content
.dbf	Attribute information
.shp	Feature geometry
.shx	Feature geometry index
.aih	Attribute index
.ain	Attribute index
.prj	Coordinate system information
.sbn	Spatial index file
.sbx	Spatial index file

File Geodatabase

A **file geodatabase** is a relational database storage format. It's a far more complex data structure than the shapefile and consists of a **.gdb** folder housing dozens of files. Its complexity renders it more versatile allowing it to store multiple feature classes and enabling topological definitions (i.e. allowing the user to define rules that govern the way different feature classes relate to one another). An example of the contents of a geodatabase is shown in the following figure.



Raster Data File Formats

Rasters are in part defined by their pixel depth. Pixel depth defines the range of distinct values the raster can store. For example, a 1-bit raster can only store 2 distinct values: 0 and 1.

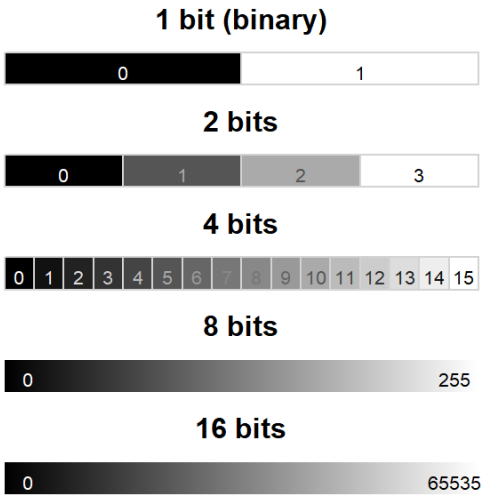


Fig. 5 Pixel depth allows for a wider range of values but also take up more space

There is a wide range of raster file formats used in the GIS world. Some of the most popular ones are listed below.

Imagine

The **Imagine** file format was originally created by an image processing software company called ERDAS. This file format consists of a single **.img** file. This is a simpler file format than the shapefile. It is sometimes accompanied by an .xml file which usually stores metadata information about the raster layer.

GeoTiff

A popular public domain raster data format is the **GeoTIFF** format. If maximum portability and platform independence is important, this file format may be a good choice.

File Geodatabase

A raster file can also be stored in a file geodatabase alongside vector files. Geodatabases have the benefit of defining image mosaic structures thus allowing the user to create “stitched” images from multiple image files stored in the geodatabase. Also, processing very large raster files can be computationally more efficient when stored in a file geodatabase as opposed to an Imagine or GeoTiff file format.

Learning Objectives

- Create a Geopandas GeoSeries and Dataframe
- Plot a basic map

Review

- Data Structures

Spatial Vector Data

Intro to GeoPandas

The goal of GeoPandas is to make working with spatial data in python easier. It combines the capabilities of pandas and shapely, providing spatial operations in pandas and a high-level interface to multiple geometries to shapely. GeoPandas enables you to easily do operations in python that would otherwise require a spatial database such as PostGIS.

Data Structures

GeoPandas implements two main data structures, a `GeoSeries` and a `GeoDataFrame`. These are subclasses of pandas Series and DataFrame, respectively.

GeoSeries

A `GeoSeries` is essentially a vector where each entry in the vector is a set of shapes corresponding to one observation. An entry may consist of only one shape (like a single polygon) or multiple shapes that are meant to be thought of as one observation (like the many polygons that make up the State of Hawaii or a country like Indonesia).

geopandas has three basic classes of geometric objects (which are actually shapely objects):

- Points / Multi-Points
- Lines / Multi-Lines
- Polygons / Multi-Polygons

```
import geopandas
from shapely.geometry import Point
s = geopandas.GeoSeries([Point(1, 1), Point(2, 2), Point(3, 3)])
s
```

```
0    POINT (1.00000 1.00000)
1    POINT (2.00000 2.00000)
2    POINT (3.00000 3.00000)
dtype: geometry
```

```
from shapely.geometry import LineString
l= geopandas.GeoSeries([LineString([Point(-77.036873,38.907192),
Point(-76.612190,39.290386,), Point(-77.408456,39.412006)])])
l
```

```
0    LINESTRING (-77.03687 38.90719, -76.61219 39.2...
dtype: geometry
```

```
from shapely.geometry import Polygon
p= geopandas.GeoSeries([Polygon([Point(-77.036873,38.907192),
Point(-76.612190,39.290386,), Point(-77.408456,39.412006)])])
p
```

```
0    POLYGON ((-77.03687 38.90719, -76.61219 39.290...
dtype: geometry
```

Note that all entries in a `GeoSeries` need not be of the same geometric type, although certain export operations will fail if this is not the case.

GeoDataFrame

A `GeoDataFrame` is a tabular data structure that contains a `GeoSeries`.

The most important property of a `GeoDataFrame` is that it always has one `GeoSeries` column that holds a special status. This `GeoSeries` is referred to as the `GeoDataFrame's` "geometry". When a spatial method is applied to a `GeoDataFrame` (or a spatial attribute like area is called), this commands will always act on the "geometry" column.

The "geometry" column – no matter its name – can be accessed through the geometry attribute (gdf.geometry), and the name of the geometry column can be found by typing gdf.geometry.name.

> **Note**
>
> A `GeoDataFrame` may also contain other columns with geometrical (shapely) objects, but only one column can be the active geometry at a time. To change which column is the active geometry column, use the `GeoDataFrame.set_geometry()` method.

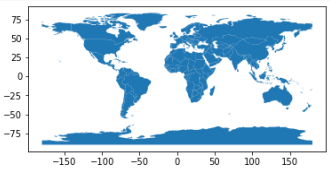
An example using the worlds `GeoDataFrame`:

```
world = geopandas.read_file(geopandas.datasets.get_path('naturalearth_lowres'))
world.head()
```

	pop_est	continent	name	iso_a3	gdp_md_est	geometry
0	920938	Oceania	Fiji	FJI	8374.0	MULTIPOLYGON (((180.00000 -16.06713, 180.00000...
1	53950935	Africa	Tanzania	TZA	150600.0	POLYGON ((33.90371 -0.95000, 34.07262 -1.05982...
2	603253	Africa	W. Sahara	ESH	906.5	POLYGON ((-8.66559 27.65643, -8.66512 27.58948...
3	35623680	North America	Canada	CAN	1674000.0	MULTIPOLYGON (((-122.84000 49.00000, -122.9742...
4	326625791	North America	United States of America	USA	18560000.0	MULTIPOLYGON (((-122.84000 49.00000, -120.0000...

```
world.plot()
```

```
<AxesSubplot:>
```



> **Learning Objectives**
>
> - Create new spatial objects (points, lines, polygons)
> - Assign the correct projection or CRS
> - Create points from a table or csv of lat and lon

> **Review**
>
> - CRS what is it?
> - Understand CRS codes
> - Vector data structures
> - Find Lat Lon of your own points, lines, polygons

Spatial Points Lines Polygons in Python

We often find ourselves in a situation where we need to generate new spatial data from scratch, or need to better understand how our data is constructed. This lesson will walk you through some of the most common forms of data generation.

```
# Import necessary modules first
import pandas as pd
import geopandas as gpd
from shapely.geometry import Point, LineString, Polygon
import fiona
import matplotlib.pyplot as plt
plt.style.use('bmh') # better for plotting geometries vs general plots.
```

Creating GeoDataFrame Geometries

A `GeoDataFrame` object is a `pandas.DataFrame` that has a column with geometry. An empty `GeoDataFrame` is just that, empty, essentially just like the pandas one. Let's create an empty `GeoDataFrame` and create a new column called geometry that will contain our Shapely objects:

```
# Create an empty geopandas GeoDataFrame
newdata = gpd.GeoDataFrame()
print(newdata)
```

```
Empty GeoDataFrame
Columns: []
Index: []
```

In order to have a working spatial dataframe we need define a few things:

GeoDataFrame Components

- data: a pandas.DataFrame, dictionary, or empty list [] containing an desired attribute data. Use [] if no data is
- crs: Coordinate Reference System of the geometry objects. Can be anything accepted by `pyproj.CRS.from_user_input()`, such as an authority string (eg "EPSG:4326") or a WKT string.
- geometry: Column name in a DataFrame to use as geometry or Shapely point, line, or polygon object.

Since geopandas takes advantage of Shapely geometric objects it is possible to create a Shapefile from a scratch by passing Shapely's geometric objects into the GeoDataFrame. This is useful as it makes it easy to convert e.g. a text file that contains coordinates into a Shapefile.

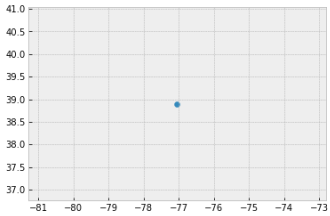
Now we have a geometry column in our GeoDataFrame but we don't have any data yet.

Create Points from list of coordinates

Creating geopandas point objects is a snap! All we need is a coordinate pair from which we generate a Shapely point geometry object, we then create a dictionary that holds that geometry and any attributes we want, and a coordinate reference system. In this case we use a ESPG code.
Click here for a more detailed explanation of this process

```
# Coordinates of the GW department of geography in Decimal Degrees
coordinate = [-77.04639494419096,  38.89934963421794]

# Create a Shapely point from a coordinate pair
point_coord = Point(coordinate)

# create a dataframe with needed attributes and required geometry column
df = {'GWU': ['Dept Geography'], 'geometry': [point_coord]}

# Convert shapely object to a geodataframe
point = gpd.GeoDataFrame(df, geometry='geometry', crs ="EPSG:4326")

# Let's see what we have
point.plot()
```

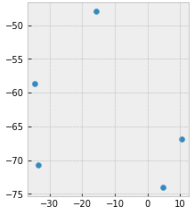
```
<AxesSubplot:>
```



We can apply the same process to a set of points stored in a pandas dataframe.

```
# list of attributes and coordinates
df = pd.DataFrame(
    {'City': ['Buenos Aires', 'Brasilia', 'Santiago', 'Bogota', 'Caracas'],
     'Country': ['Argentina', 'Brazil', 'Chile', 'Colombia', 'Venezuela'],
     'lat': [-34.58, -15.78, -33.45, 4.60, 10.48],
     'lon': [-58.66, -47.91, -70.66, -74.08, -66.86]})

# Create a Shapely points from the coordinate-tuple list
ply_coord = [Point(x, y) for x, y in zip(df.lat, df.lon)]

# Convert shapely object to a geodataframe with a crs
poly = gpd.GeoDataFrame(df, geometry=ply_coord, crs ="EPSG:4326")

# Let's see what we have
poly.plot()
```

```
<AxesSubplot:>
```



adapted from geopandas

Creating Points from csv of latitude and longitude (lat, lon)

One of the most common data creation tasks is creating a shapefile from a list of points or a `.csv` file. Luckily getting this data into usable format is easy enough.

First we have to create an example `.csv` dataset to work from:

```
import pandas as pd
# create an outline of Washington DC and write to csv
path_to_csv = r'../temp/points.csv'
points = {'Corner':['N','E','S','W'],
          'lon': [-77.0412826538086, -77.11681365966797, -77.01896667480469,
-77.0412826538086],
          'lat': [38.99570671505043, 38.936713143230044, 38.807610542357594,
38.99570671505043]}
points = pd.DataFrame.from_dict(points)
points.to_csv(path_to_csv)
```

To create a `geodataframe` from our data you simply need to read it back in, an specify the geometry column values using `points_from_xy` pointing it to the correct columns of `df`, namely `df.lon` anf `df.lat`.

```
# read the point data in
df = pd.read_csv(path_to_csv)

# Create a geodataframe from the data using and 'EPSG' code to assign WGS84
coordinate reference system
points= gpd.GeoDataFrame(df, geometry=gpd.points_from_xy(x=df.lon, y=df.lat), crs
= 'EPSG:4326')
points
```

	Unnamed: 0	Corner	lon	lat	geometry
0	0	N	-77.041283	38.995707	POINT (-77.04128 38.99571)
1	1	E	-77.116814	38.936713	POINT (-77.11681 38.93671)
2	2	S	-77.018967	38.807611	POINT (-77.01897 38.80761)
3	3	W	-77.041283	38.995707	POINT (-77.04128 38.99571)

In this case `points_from_xy()` was used to transform lat and lon into a list of `shapely.Point` objects. This then is used as the geometry for the GeoDataFrame. (`points_from_xy()` is simply an enhanced wrapper for `[Point(x, y) for x, y in zip(df.lon, df.lat)]`)

> **Tip**
>
> - Although we say “lat lon” python uses “lon lat” instead, this follows the preference for using x,y for notation.
> - Typically, like the data above, these data are stored in WGS84 lat lon, but be sure to check this, another common format is UTM coordinates (look for values around 500,000 east to west and measured in meters)

Creating Spatial lines

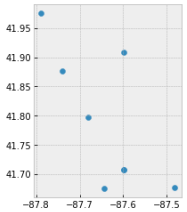
Following the examples above we can specify lines easily. In this case let’s say we have lines tracking three people riding their bikes through town. We keep track of their unique id `ID`, their location `X,Y`, and their `Speed`, and read in the data below:

```
from io import StringIO
data = """
ID,X,Y,Speed
1,  -87.789,  41.976,  16
1,  -87.482,  41.677,  17
2,  -87.739,  41.876,  16
2,  -87.681,  41.798,  16
2,  -87.599,  41.708,  16
3,  -87.599,  41.908,  17
3,  -87.598,  41.708,  17
3,  -87.643,  41.675,  17
"""
# use StringIO to read in text chunk
df = pd.read_table(StringIO(data), sep=',')
```

Let’s convert these to points and take a look. Notice that points are not a good replacement for lines in the case, we have three individuals, and they need to be treated separately.

```
#zip the coordinates into a point object and convert to a GeoData Frame
points = [Point(xy) for xy in zip(df.X, df.Y)]
geo_df = gpd.GeoDataFrame(df, geometry=points, crs = 'EPSG:4326')
geo_df.plot()
```

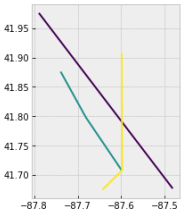
```
<AxesSubplot:>
```



Now let’s tread these data as lines, we can take advantage of the column `ID` to `.groupby`. Luckily geopandas `.groupby` is consistent with the use in pandas. So here we `.groupby(['ID'])`, for each `ID` group we convert the values to a list, and store it in a Fiona `LineString` object.

```
# treat each `ID` group of points as a line
lines = geo_df.groupby(['ID'])['geometry'].apply(lambda x:
LineString(x.tolist()))

# store as a GeodataFrame and add 'ID' as a column (currently stored as the
'index')
lines = gpd.GeoDataFrame(lines, geometry='geometry', crs="EPSG:4326")
lines.reset_index(inplace=True)
lines.plot(column='ID')
```

```
<AxesSubplot:>
```



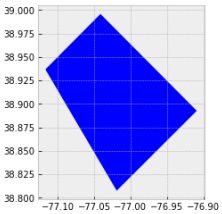
Now we can see that each line is treated separately by `ID`, and plot them using `.plot(column='ID')`.

Creating Spatial Polygons

Creating a polyon in geopandas is very similiar to the other exercises. First we create a Fiona geometry object from our coordinates, add that to a dataframe with any attributes and then create a `GeoDataFrame` with an assigned coordinate reference system (CRS).

```
# list of coordindate pairs
coordinates = [[ -77.0412826538086, 38.99570671505043 ], [ -77.11681365966797,
38.936713143230044 ], [ -77.01896667480469, 38.807610542357594],
                [-76.90910339355469,  38.892636142310295]]

# Create a Shapely polygon from the coordinate-tuple list
ply_coord = Polygon(coordinates)

# create a dictionary with needed attributes and required geometry column
df = {'Attribute': ['name1'], 'geometry': ply_coord}

# Convert shapely object to a geodataframe
poly = gpd.GeoDataFrame(df, geometry='geometry', crs ="EPSG:4326")

# Let's see what we have
poly.plot()
```

```
<AxesSubplot:>
```



Creating Spatial Points (admittedly the long way)

Since geopandas takes advantage of Shapely geometric objects it is possible to create a Shapefile from a scratch by passing Shapely's geometric objects into the GeoDataFrame. This is useful as it makes it easy to convert e.g. a text file that contains coordinates into a Shapefile.

Let's create an empty `GeoDataFrame` and create a new column called geometry that will contain our Shapely objects:

```
# Create an empty geopandas GeoDataFrame
newdata = gpd.GeoDataFrame()
# Create a new column called 'geometry' to the GeoDataFrame
newdata['geometry'] = None

print(newdata)
```

```
Empty GeoDataFrame
Columns: [geometry]
Index: []
```

Let's create a Shapely Point representing the GWU Department of Geography that we can insert to our GeoDataFrame:

```
# Coordinates of the GW department of geography in Decimal Degrees
coordinates = (-77.04639494419096,  38.89934963421794)

# Create a Shapely polygon from the coordinate-tuple list
point = Point(coordinates)

# Let's see what we have
point
```

Okay, so now we have appropriate Polygon -object.

Let's insert the polygon into our 'geometry' column in our GeoDataFrame:

```
# Insert the polygon into 'geometry' -column at index 0
newdata.loc[0, 'geometry'] = point

# Let's see what we have now
newdata
```

	geometry
0	POINT (-77.04639 38.89935)

Now we have a GeoDataFrame with Point that we can export to a Shapefile.

Let's add another column to our GeoDataFrame called Location with text GWU Geography.

```
# Add a new column and insert data
newdata.loc[0, 'Location'] = 'GWU Geography'

# Let's check the data
newdata
```

	geometry	Location
0	POINT (-77.04639 38.89935)	GWU Geography

Okay, now we have additional information that is useful to be able to recognize what the feature represents.

Before exporting the data it is useful to determine the coordinate reference system (CRS, 'projection') for the GeoDataFrame.

GeoDataFrame has a property called `.crs` that (review here) shows the coordinate system of the data which is empty (None) in our case since we are creating the data from the scratch (e.g. `newdata.crs` returns `None`).

Let's add a crs for our GeoDataFrame. A Python module called fiona has a nice function called from_epsg() for passing coordinate system for the GeoDataFrame. Next we will use that and determine the projection to WGS84 (epsg code: 4326) which is the most common choice for lat lon CRSs:

```
# Import specific function 'from_epsg' from fiona module
from fiona.crs import from_epsg

# Set the GeoDataFrame's coordinate system to WGS84
newdata.crs = from_epsg(code = 4326)

# Let's see how the crs definition looks like
newdata.crs
```

```
/home/mmann1123/anaconda3/envs/pygisbookgw/lib/python3.7/site-
packages/pyproj/crs/crs.py:68: FutureWarning: '+init=<authority>:<code>' syntax is
deprecated. '<authority>:<code>' is the preferred initialization method. When
making the change, be mindful of axis order changes:
https://pyproj4.github.io/pyproj/stable/gotchas.html#axis-order-changes-in-proj-6
  return _prepare_from_string(" ".join(pjargs))
```

```
<Geographic 2D CRS: +init=epsg:4326 +no_defs +type=crs>
Name: WGS 84
Axis Info [ellipsoidal]:
- lon[east]: Longitude (degree)
- lat[north]: Latitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

Finally, we can export the data using GeoDataFrames .to_file() -function. The function works similarly as numpy or pandas, but here we only need to provide the output path for the Shapefile. Easy isn't it!:

```
# Determine the output path for the Shapefile
outfp = r"../temp/gwu_geog.shp"

# Write the data into that Shapefile
newdata.to_file(outfp)
```

Tip

Because GeoPandas are so intertwined spend the time to learn more about here Pandas User Guide

---

Adapted from Intro to Python GIS

Learning Objectives

- Create new raster objects
- Assign the correct projection or CRS

Review

- Please review Affine transformation

Spatial Raster Data in Python

A raster data model uses an array of cells, or pixels, to represent real-world objects. Raster datasets are commonly used for representing and managing imagery, surface temperatures, digital elevation models, and numerous other entities. A raster can be thought of as a special case of an area object where the area is divided into a regular grid of cells. But a regularly spaced array of marked points may be a better analogy since rasters are stored as an array of values where each cell is defined by a single coordinate pair inside of most GIS environments. Implicit in a raster data model is a value associated with each cell or pixel. This is in contrast to a vector model that may or may not have a value associated with the geometric primitive.

In order to work with raster data we will be using `rasterio` and later `geowombat`. Behind the scenes a `numpy.ndarray` does all the heavy lifting. To understand how raster works it helps to construct one from scratch.

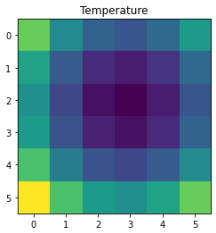
Here we create two `ndarray` objects one `X` spans [-90°,90°] longitude, and `Y` covers [-90°,90°] latitude.

```
import numpy as np
x = np.linspace(-90, 90, 6)
y = np.linspace(90, -90, 6)
X, Y = np.meshgrid(x, y)
X
```

```
array([[-90., -54., -18.,  18.,  54.,  90.],
       [-90., -54., -18.,  18.,  54.,  90.],
       [-90., -54., -18.,  18.,  54.,  90.],
       [-90., -54., -18.,  18.,  54.,  90.],
       [-90., -54., -18.,  18.,  54.,  90.],
       [-90., -54., -18.,  18.,  54.,  90.]])
```

Let's generate some data representing temperature and store it an array `z`

```
import matplotlib.pyplot as plt

Z1 =  np.abs(((X - 10) ** 2 + (Y - 10) ** 2) / 1 ** 2)
Z2 =  np.abs(((X + 10) ** 2 + (Y + 10) ** 2) / 2.5 ** 2)
Z =  (Z1 - Z2)

plt.imshow(Z)
plt.title("Temperature")
plt.show()
```



Note that *z contains no data on its location*. Its just an array, the information stored in `x` and `y` aren't associated with it at all. This location data will often be stored in the header of file. In order to 'locate' the array on the map we will use affine transformations.

Assigning spatial data to an array in python

Ok we have an array of data and some coordinates for each cell, but how do we create a spatial dataset from it? In order to do this we need three components:

1. An array of data and typically the xy coordinates
2. A coordinate reference system which defines what coordinate space we are using (e.g. degrees or meters, where is the prime meridian etc)
3. A transform defining the coordinate of the upper left hand corner and the cell resolution

> **Note**
>
> These topic is covered extensively in the next chapter. We will briefly introduce the topic here, but will go into the details later.
>
> - For more info on transforms go here.
> - For more info on coordinate references systems go here
> - or to understand CRS codes go here.
> - To help bring it all together, read/writing rasters go to Reading & Writing Rasters with Rasterio.

Once you have those components you can write out a working spatial raster dataset in python in a few lines of code. We just need to provide the information listed above in a format that rasterio understands.

```
from rasterio.transform import Affine
import rasterio as rio

res = (x[-1] - x[0]) / 240.0
transform = Affine.translation(x[0] - res / 2, y[0] - res / 2) * Affine.scale(res,
res)

# open in 'write' mode, unpack profile info to dst
with rio.open(
   "../temp/new_raster.tif",
   "w",
   driver="GTiff",          # output file type
   height=Z.shape[0],       # shape of array
   width=Z.shape[1],
   count=1,                 # number of bands
   dtype=Z.dtype,           # output datatype
   crs="+proj=latlong",     # CRS
   transform=transform,     # location and resolution of upper left cell
) as dst:
   # check for number of bands
   if dst.count == 1:
       # write single band
       dst.write(Z, 1)
   else:
       # write each band individually
       for band in range(len(Z)):
           # write data, band # (starting from 1)
           dst.write(Z[band], band + 1)
```

> **Note**
>
> Raster data is often 'multiband' (e.g. red, green, blue), so I provided code that works for both multiband and single band data.
>
> If you are storing multiband data the dimensions should be stored as (band, y, x).
>
> Read more about this here: Reading & Writing Rasters with Rasterio

---

> **Learning Objectives**
>
> - Learn what a Coordinate Reference System (CRS) is
> - Learn about the properties of a CRS
> - Differentiate projected and geographic CRSs
> - Understand CRS impact on shape, area and distance

> **Review**
>
> - Affine Transforms

What is a CRS?

Implicit with any GIS data is a spatial reference system. It can consist of a simple arbitrary reference system such as a 10 m x 10 m sampling grid in a wood lot or, the boundaries of a soccer field or, it can consist of a geographic reference system, i.e. one where the spatial features are mapped to an earth based reference system. The focus of this topic is on earth reference systems which can be based on a Geographic Coordinate System (GCS) or a Project Coordinate System (PCS).

Geographic Coordinate Systems

A geographic coordinate system is a reference system for identifying locations on the curved surface of the earth. Locations on the earth's surface are measured in angular units from the center of the earth relative to two planes: the plane defined by the equator and the plane defined by the prime meridian (which crosses Greenwich England). A location is therefore defined by two values: a latitudinal value and a longitudinal value.

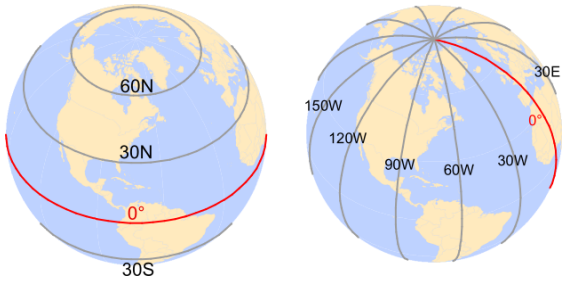
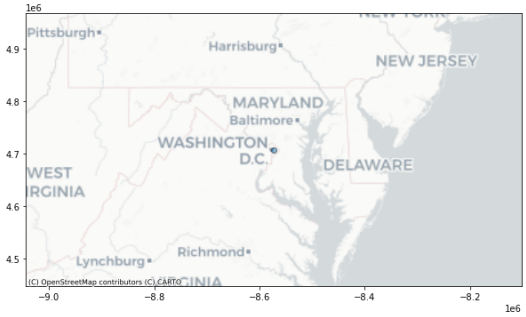


Fig. 6 Examples of latitudinal lines are shown on the left and examples of longitudinal lines are shown on the right. The 0° degree reference lines for each are shown in red (equator for latitudinal measurements and prime meridian for longitudinal measurements).

A latitude measures the angle from the equatorial plane to the location on the earth's surface. A longitude measures the angle between the prime meridian plane and the north-south plane that intersects the location of interest. For example The George Washington University is located at around 38.89° North and -77.04° West. In a GIS system, the North-South and East-West directions are encoded as signs. North and East are assigned a positive (+) sign and South and West are assigned a negative (-) sign. The university location is therefore encoded as +38.89° and -77.04°.

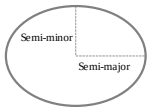
```
import geopandas
import contextily as ctx
cities = geopandas.read_file(geopandas.datasets.get_path('naturalearth_cities'))
cities = cities[cities.name == 'Washington, D.C.']
cities = cities.to_crs(epsg=3857) # project to webmercator

ax = cities.plot(figsize=(10, 10), alpha=0.5, edgecolor='k')
ctx.add_basemap(ax, source=ctx.providers.CartoDB.Positron,zoom = 6, reset_extent =
True)
```



A GCS is defined by an **ellipsoid**, **geoid** and **datum**. These elements are presented next.

Sphere and Ellipsoid

Assuming that the earth is a perfect sphere greatly simplifies mathematical calculations and works well for small-scale maps (maps that show a *large* area of the earth). However, when working at larger scales, an ellipsoid representation of earth may be desired if accurate measurements are needed. An ellipsoid is defined by two radii: the semi-major axis (the equatorial radius) and the semi-minor axis (the polar radius).



The reason the earth has a slightly ellipsoidal shape has to do with its rotation which induces a centripetal force along the equator. This results in an equatorial axis that is roughly 21 km longer than the polar axis.

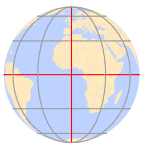


Fig. 7 The earth can be mathematically modeled as a simple sphere (left) or an ellipsoid (right).

Our estimate of these radii is quite precise thanks to satellite and computational capabilities. The semi-major axis is 6,378,137 meters and the semi-minor axis is 6,356,752 meters.

Differences in distance measurements along the surfaces of an ellipsoid vs. a sphere are small but measurable (the difference can be as high as 20 km) as illustrated in the following lattice plots.

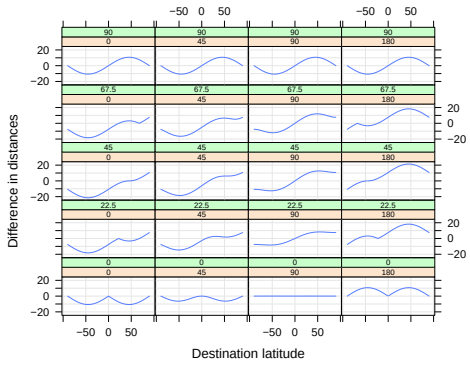


Fig. 8 Differences in distance measurements between the surface of a sphere and an ellipsoid. Each graphic plots the differences in distance measurements made from a single point location along the 0° meridian identified by the green colored box (latitude value) to various latitudinal locations along a longitude (whose value is listed in the bisque colored box). For example, the second plot from the top-left corner plot shows the differences in distance measurements made from a location at 90° north (along the prime meridian) to a range of latitudinal locations along the 45° meridian.

Geoid

Representing the earth's true shape, the *geoid*, as a mathematical model is crucial for a GIS environment. However, the earth's shape is not a perfectly smooth surface. It has undulations resulting from changes in gravitational pull across its surface. These undulations may not be visible with the naked eye, but they are measurable and can influence locational measurements. Note that we are not including mountains and ocean bottoms in our discussion, instead we are focusing solely on the earth's gravitational potential which can be best visualized by imagining the earth's surface completely immersed in water and measuring the distance from the earth's center to the water surface over the entire earth surface.

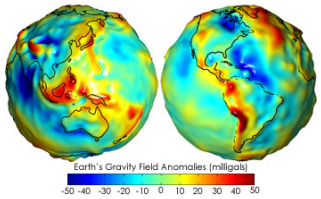


Fig. 9 Earth's geoid with gravitational field shown in rainbow colors. The ondulations depicted in the graphics are exaggerated for visual effects. (source- NASA)

The earth's gravitational field is dynamic and is tied to the flow of the earth's hot and fluid core. Hence its geoid is constantly changing, albeit at a large temporal scale.The measurement and representation of the earth's shape is at the heart of geodesy–a branch of applied mathematics.

Datum

So how are we to reconcile our need to work with a (simple) mathematical model of the earth's shape with the ondulating nature of the earth's surface (i.e. its geoid)? The solution is to align the geoid with the ellipsoid (or sphere) representation of the earth and to map the earth's surface features onto this ellipsoid/sphere. The alignment can be local where the ellipsoid surface is closely fit to the geoid at a particular location on the earth's surface (such as the state of Kansas) or **geocentric** where the ellipsoid is aligned with the center of the earth. How one chooses to align the ellipsoid to the geoid defines a **datum**.

Local Datum

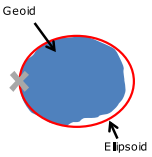


Fig. 10 A local datum couples a geoid with the ellipsoid at a location on each element's surface.

There are many local datums to choose from, some are old while others are more recently defined. The choice of datum is largely driven by the location of interest. For example, when working in the US, a popular local datum to choose from is the North American Datum of 1927 (or NAD27 for short). NAD27 works well for the US but it's not well suited for other parts of the world. For example, a far better local datum for Europe is the European Datum of 1950 (ED50 for short). Examples of common local datums are shown in the following table:

Local datum	Acronym	Best for	Comment
North American Datum of 1927	NAD27	Continental US	This is an old datum but still prevalent because of the wide use of older maps.
European Datum of 1950	ED50	Western Europe	Developed after World War II and still quite popular today. Not used in the UK.
World Geodetic System 1972	WGS72	Global	Developed by the Department of Defense.

Geocentric Datum

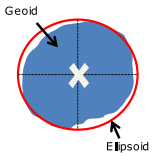


Fig. 11 A geocentric datum couples a geoid with the ellipsoid at each element's center of mass.

Many of the modern datums use a geocentric alignment. These include the popular World Geodetic Survey for 1984 (WGS84) and the North American Datums of 1983 (NAD83). Most of the popular geocentric datums use the WGS84 ellipsoid or the GRS80 ellipsoid. These two ellipsoids share nearly identical semi-major and semi-minor axes: 6,378,137 meters and 6,356,752 meters respectively. Examples of popular geocentric datums are shown in the following table:

Geocentric datum	Acronym	Best for	Comment
North American Datum of 1983	NAD83	Continental US	This is one of the most popular modern datums for the contiguous US.
European Terrestrial Reference System 1989	ETRS89	Western Europe	This is the most popular modern datum for much of Europe.
World Geodetic System 1984	WGS84	Global	Developed by the Department of Defense.

Building the Geographic Coordinate System

A Geographic Coordinate System (GCS) is defined by the ellipsoid model and by the way this ellipsoid is aligned with the geoid (defining the datum). It is important to know which GCS is associated with a GIS file or a map document reference system. This is particularly true when the overlapping layers are tied to different datums (and therefore GCS'). This is because a location on the earth's surface can take on different coordinate values. For example, a location recorded in an NAD 1927 GCS having a coordinate pair of 44.56698° north and 69.65939° west will register a coordinate value of 44.56704° north and 69.65888° west in a NAD83 GCS and a coordinate value of 44.37465° north and -69.65888° west in a sphere based WGS84 GCS. If the coordinate systems for these point coordinate values were not properly defined, then they could be misplaced on a map. This is analogous to recording temperature using different units of measure (degrees Celsius, Fahrenheit and Kelvin)–each unit of measure will produce a different numeric value.

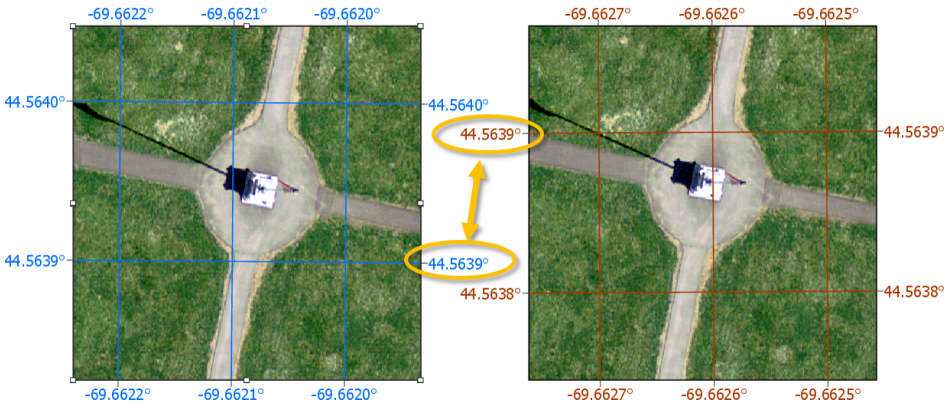


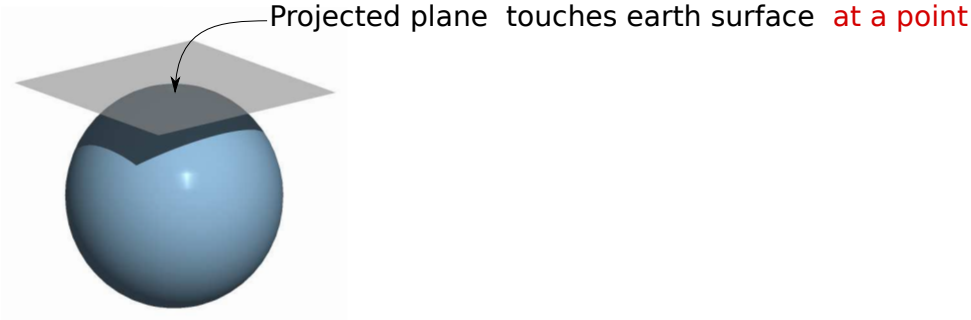
Fig. 12 Map of the Colby flagpole in two different geographic coordinate systems (*GCS NAD 1983* on the left and *GCS NAD 1927* on the right). Note the offset in the 44.5639° line of latitude relative to the flagpole. Also note the 0.0005° longitudinal offset between both reference systems.

Projected Coordinate Systems

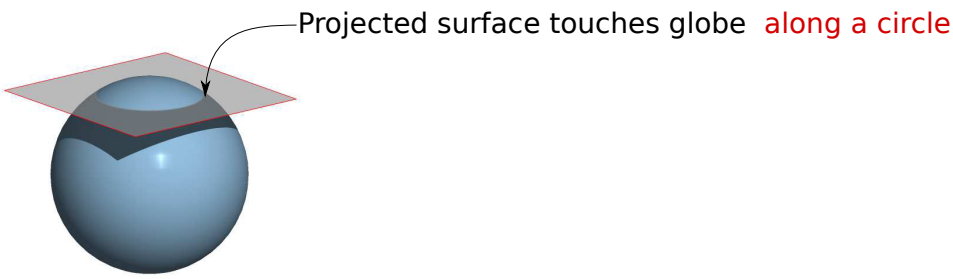
The surface of the earth is curved but maps are flat. A projected coordinate system (PCS) is a reference system for identifying locations and measuring features on a flat (map) surface. It consists of lines that intersect at right angles, forming a grid. Projected coordinate systems (which are based on Cartesian coordinates) have an origin, an *x* axis, a *y* axis, and a linear unit of measure. Going from a GCS to a PCS requires mathematical transformations. The myriad of projection types can be aggregated into three groups: **planar**, **cylindrical** and **conical**.

Planar Projections

Tangent Case A planar projection (aka Azimuthal projection) maps the earth surface features to a flat surface that touches the earth's surface at a point (**tangent** case),



Secant Case or along a line of tangency (a **secant** case).



This projection is often used in mapping polar regions but can be used for any location on the earth's surface (in which case they are called oblique planar projections).

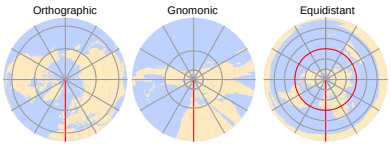
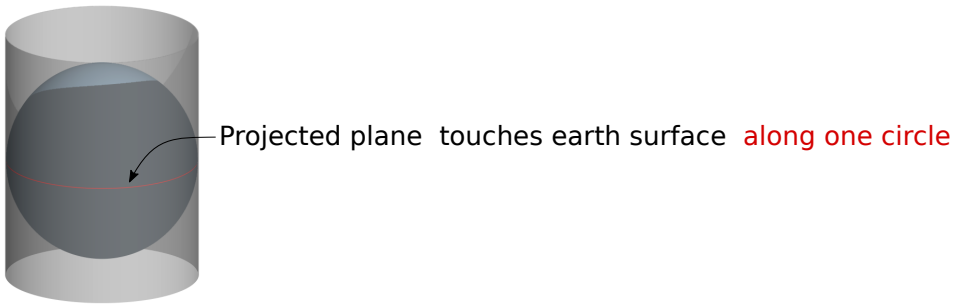


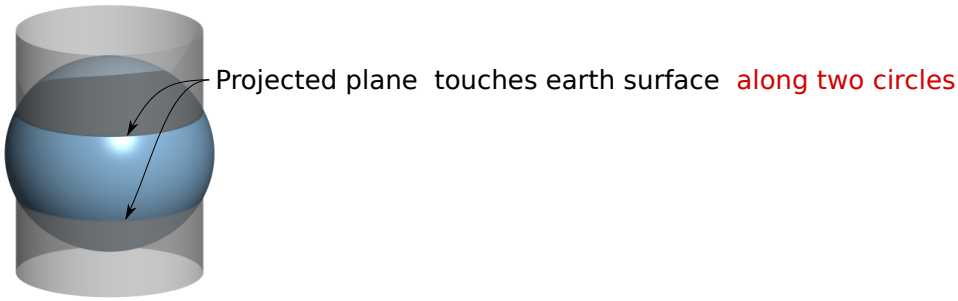
Fig. 13 Examples of three planar projections - orthographic (left), gnomonic (center) and equidistant (right). Each covers a different spatial range (with the latter covering both northern and southern hemispheres) and each preserves a unique set of spatial properties.

Cylindrical Projection

A cylindrical map projection maps the earth surface onto a map rolled into a cylinder (which can then be flattened into a plane). The cylinder can touch the surface of the earth along a single line of tangency (a **tangent** case),



or along two lines of tangency (a **secant** case).



The cylinder can be tangent to the equator or it can be oblique. A special case is the Transverse aspect which is tangent to lines of longitude. This is a popular projection used in defining the Universal Transverse Mercator (UTM) and State Plane coordinate systems. The UTM PCS covers the entire globe and is a popular coordinate system in the US. It's important to note that the UTM PCS is broken down into zones and therefore limits its extent to these zones that are 6° wide. For example, the State of Maine (USA) uses the UTM coordinate system (Zone 19 North) for most of its statewide GIS maps. Most USGS quad maps are also presented in a UTM coordinate system. Popular datums tied to the UTM coordinate system in the US include NAD27 and NAD83. There is also a WGS84 based UTM coordinate system.

Distortion is minimized along the tangent or secant lines and increases as the distance from these lines increases.

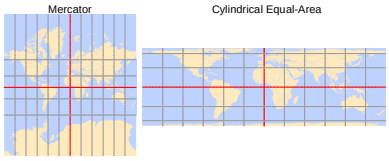
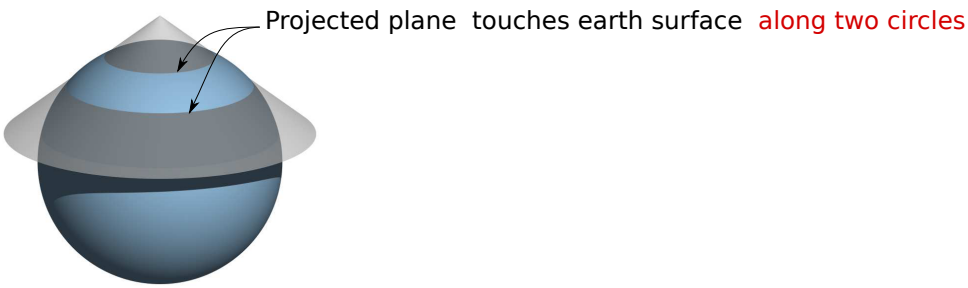


Fig. 14 Examples of two cylindrical projections Mercator (preserves shape but distorts area and distance) and equal-area (preserves area but distorts shape).```

or along two lines of tangency (a **secant** case).



Distortion is minimized along the tangent or secant lines and increases as the distance from these lines increases. When distance or area measurements are needed for the contiguous 48 states, use one of the conical projections such as Equidistant Conic (distance preserving) or Albers Equal Area Conic (area preserving).

Conical projections are also popular PCS' in European maps such as Europe Albers Equal Area Conic and Europe Lambert Conformal Conic.

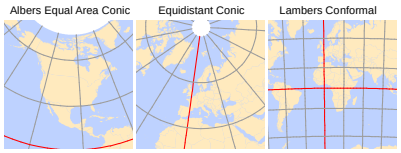
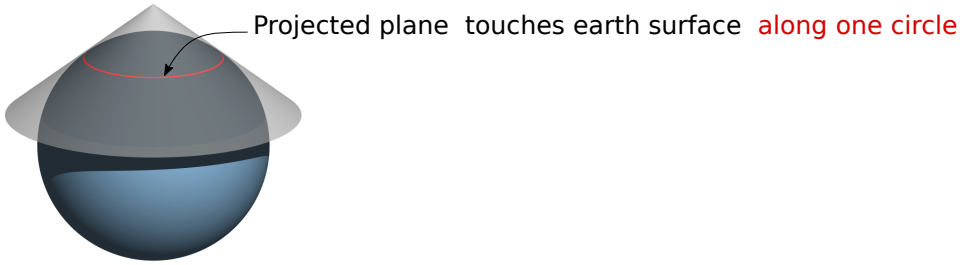


Fig. 15 Examples of three conical projections - Albers equal area (preserves area), equidistant (preserves distance) and conformal (preserves shape).

Conical Projection

A conical map projection maps the earth surface onto a map rolled into a cone. Like the cylindrical projection, the cone can touch the surface of the earth along a single line of tangency (a **tangent** case),



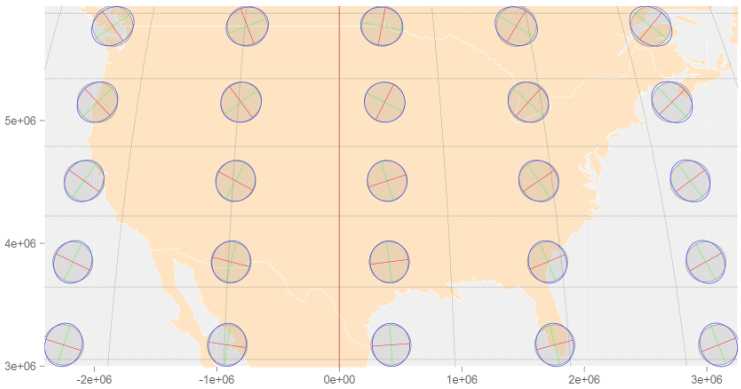
Spatial Properties

All projections distort real-world geographic features to some degree. The four spatial properties that are subject to distortion are: **shape**, **area**, **distance** and **direction**. A map that preserves shape is called conformal; one that preserves area is called equal-area; one that preserves distance is called equidistant; and one that preserves direction is called azimuthal.

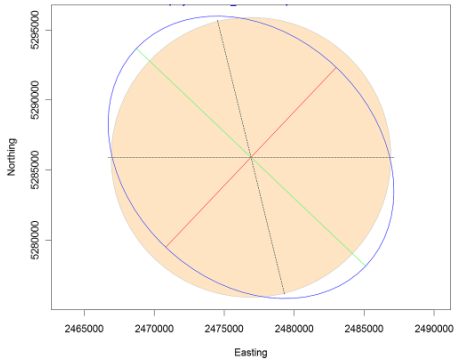
> For most GIS applications (e.g. ArcGIS and QGIS), many of the built-in projections are named after the spatial properties they preserve.

Each map projection is good at preserving only one or two of the four spatial properties. So when working with small-scale (large area) maps and when multiple spatial properties are to be preserved, it is best to break the analyses across different projections to minimize errors associated with spatial distortion.

If you want to assess a projection's spatial distortion across your study region, you can generate Tissot indicatrix (TI) ellipses. The idea is to project a small circle (i.e. small enough so that the distortion remains relatively uniform across the circle's extent) and to measure its distorted shape on the projected map. For example, in assessing the type of distortion one could expect with a Mollweide projection across the continental US, a grid of circles could be generated at regular latitudinal and longitudinal intervals.



Note the varying levels of distortion type and magnitude across the region. Let's explore a Tissot circle at 44.5°N and 69.5°W (near Waterville Maine):



The plot shows a perfect circle (displayed in a filled bisque color) that one would expect to see if no distortion was at play. The blue distorted ellipse (the indicatrix) is the transformed circle for this particular projection and location. The green and red lines show the magnitude and direction of the ellipse's major and minor axes respectively. These lines can also be used to assess scale distortion (note that scale distortion can vary as a function of bearing). The green line shows maximum scale distortion and the red line shows minimum scale distortion–these are sometimes referred to as the principal directions. In this working example, the principal directions are 1.1293 and 0.8856. A scale value of 1 indicates no distortion. A value less than 1 indicates a smaller-than-true scale and a value greater than 1 indicates a greater-than-true scale.

Projections can distort scale, but this does not necessarily mean that area is distorted. In fact, for this particular projection, area is relatively well preserved despite distortion in principal directions. Area distortion can easily be computed by taking the product of the two aforementioned principal directions. In this working example, area distortion is 1.0001 (i.e. negligible).

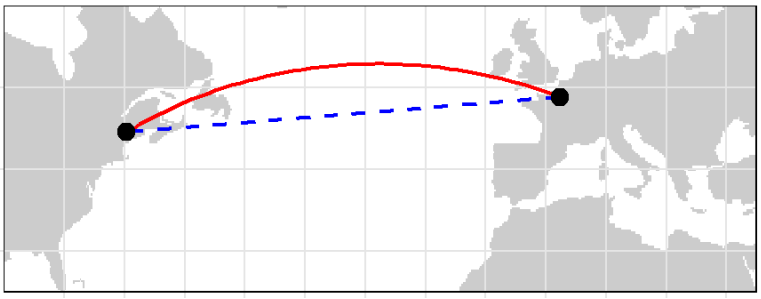
The north-south dashed line in the graphic shows the orientation of the meridian. The east-west dotted line shows the orientation of the parallel.

> It's important to recall that these distortions occur at the point where the TI is centered and not necessarily across the region covered by the TI circle.

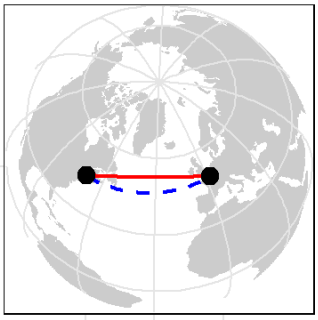
Geodesic geometries

The reason projected coordinate systems introduce errors in their geometric measurements has to do with the nature of the projection whereby the distance between two points on a sphere or ellipsoid will be difficult to replicate on a projected coordinate system unless these points are relatively close to one another. In most cases, such errors can be tolerated if the expected level of precision is met; many other sources of error in the spatial representation of the features can often usurp any measurement errors made in a projected coordinate system. However, if the scale of analysis is small (i.e. the spatial extent covers a large proportion of the earth's surface such as the North American continent), then the measurement errors associated with a projected coordinate system may no longer be acceptable. A way to circumvent projected coordinate system limitations is to adopt a geodesic solution. A **geodesic distance** is the shortest distance between two points on an ellipsoid (or spheroid). Likewise, a **geodesic area** measurement is one that is measured on an ellipsoid. Such measurements are independent of the underlying projected coordinate system. The Tissot circles presented in figures from the last section were all generated using geodesic geometry.

If you are not convinced of the benefits afforded by geodesic geometry, compare the distances measured between two points located on either sides of the Atlantic in the following map. The blue dashed line represents the shortest distance between the two points on a *planar* coordinate system. The red line represents the shortest distance between the two points as measured on a *spheroid*.



At first glance, the geodesic distance may seem nonsensical given its curved appearance on the projected map. However, this curvature is a byproduct of the current reference system's increasing distance distortion as one progresses poleward. If you are still not convinced, you can display the geodesic and planar distance layers on a 3D globe (or a projection that mimics the view of the 3D earth as viewed from space centered on the mid-point of the geodesic line segment).



So if a geodesic measurement is more precise than a planar measurement, why not perform all spatial operations using geodesic geometry? In many cases, a geodesic approach to spatial operations can be perfectly acceptable and is even encouraged. The downside is in its computational requirements. It's far more computationally efficient to compute area/distance on a plane than it is on a spheroid. This is because geodesic calculations have no simple algebraic solutions and involve approximations that may require iterative solutions. So this may be a computationally taxing approach if processing millions of line segments.

Note that not all geodesic measurement implementations are equal. Some more efficient algorithms that minimize computation time may reduce precision in the process. Some of ArcMap's functions offer the option to compute geodesic distances and areas however ArcMap does not clearly indicate how its geodesic calculations are implemented. The data analysis environment R has a package called `geosphere` that implements well defined geodesic measurement algorithms adopted from the authoritative set of GeographicLib libraries.

Understanding a CRS: Proj4 and CRS codes

Learning Objectives

- Understand how to read a `PROJ.4` string
- Understand how a `PROJ.4` string relates to an `EPSG` code
- Visually explore changing `PROJ.4` string parameters

Review

- CRS what is it?

By: Steven Chao

Intro to Coordinate Reference Systems

A coordinate reference system tells Python where and how to place coordinates on Earth's surface. For a detailed lesson on projections please go here.

As you may recall, a coordinate reference system can either be a geographic coordinate system or a projected coordinate system (also known as a map projection). A geographic coordinate system consists of a datum, which consists of an ellipsoid model and how this ellipsoid is aligned with the geoid. It's spherical in nature, and units are typically angular (latitude and longitude). A projected coordinate system is created by taking a geographic coordinate system and using math to transform a 3-D surface onto a flat, 2-D surface. Units are typically linear and are oftentimes measured in meters. For a refresher on coordinate systems, check out this YouTube video.

All map projections introduce error because they are inherently imperfect. The "best" or "perfect" projection to use is highly dependent on what needs to be mapped and where. Therefore, as you can expect, there are many projections to choose from.

In this chapter, we will check, utilize, change, and create CRSs in Python. To provide visual examples along the way, we will use a shapefile of the Washington, DC, boundary from Open Data DC (https://opendata.dc.gov/datasets/washington-dc-boundary).

Let's begin by reading in this dataset (no need to download the shapefile as we will pull it directly from the website).

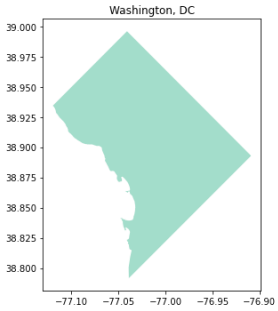
```
# Import module
import geopandas as gpd

# Read data (shapefile)
dc =
gpd.read_file("https://opendata.arcgis.com/datasets/7241f6d500b44288ad983f0942b39663_10.geojson")
```

Let's also define a function that will allow us to quickly produce a map of our data.

```
# Import module
import matplotlib.pyplot as plt

def map_data(data, header):
    '''Function superimposes all the data on a map and sets a title for the
map.'''

    # Create subplots
    fig, ax = plt.subplots(figsize = (10, 5))

    # Set colors
    colors = ["#a3ddcb", "#03506f"]

    # Iterate through list of data and colors to superimpose them onto map
    for i in range(0, len(data)):
        data[i].plot(facecolor = colors[i], ax = ax)

    # Add title
    plt.title(header)

    # Utilize BMH plotting style
    plt.style.use("bmh")

    # Remove empty white space around the plot
    plt.tight_layout()
```

Now, we can see how the shapefile we just read in looks like.

```
# Create map using DC shapefile
map_data([dc], "Washington, DC")
```



It looks like Washington, DC, albeit a bit squished! Next, let's see how we can work with CRSs in Python.

Checking and Setting Coordinate Reference Systems in Python

Checking a dataset's coordinate reference system

To check the dataset's CRS (assuming it has one), use the `.crs` attribute.

```
# Get CRS for DC shapefile
print("CRS: {}".format(dc.crs))
```

```
CRS: epsg:4326
```

For our shapefile, the output is a dictionary with a value of `'epsg:4326'`. We'll cover what this means in more detail later in this chapter, but just know for now that an `EPSG` code is a way to reference a CRS.

Change a dataset's CRS

To change a dataset's CRS, we'll need to reproject the data. Here, we will project copies of our data twice using the `to_crs()` function, once with a `PROJ.4` string and once with an `EPSG` code. Both of these values reference the same CRS (in this case, NAD83)–they are simply different ways to reference it. The projection will remain in latitude and longitude, but we will change the ellipsoid and datum.

```
# Example 1: Create a copy of the DC shapefile
dc_reproject_proj4 = dc.copy()

# Example 1: Reproject the data to NAD83 using PROJ.4 string
# Source: https://spatialreference.org/ref/epsg/nad83/
dc_reproject_proj4 = dc_reproject_proj4.to_crs("+proj=longlat +ellps=GRS80
+datum=NAD83 +no_defs")


# Example 2: Create a copy of the DC shapefile
dc_reproject_epsg = dc.copy()

# Example 2: Reproject the data to NAD83 using EPSG code
dc_reproject_epsg = dc_reproject_epsg.to_crs(epsg=4269)
```

There are some other formats we can pass as values, but we'll cover `PROJ.4` string and `EPSG` in this chapter.

When we call the `.crs` attribute, it's no longer `'epsg:4326'`, which means that the data has been reprojected!

```
# Example 1: Print new CRS of DC
print("Example 1 (PROJ.4 string) CRS: {}".format(dc_reproject_proj4.crs))

# Example 2: Print new CRS of DC
print("Example 2 (EPSG code) CRS: {}".format(dc_reproject_epsg.crs))
```

```
Example 1 (PROJ.4 string) CRS: +proj=longlat +ellps=GRS80 +datum=NAD83 +no_defs
+type=crs
Example 2 (EPSG code) CRS: epsg:4269
```

PROJ.4 String

A `PROJ.4` string identifies and defines a particular CRS. The string holds the parameters (e.g., units, datum) of a given CRS.

Sources: Using PROJ; Lesson 4. Understand EPSG, WKT and Other CRS Definition Styles, Leah Wasser

What is in a PROJ.4 string?

Multiple parameters are needed in the string to describe a CRS. To separate the parameters in the string and identify each individual parameter, each parameter begins with a `+` sign. A CRS parameter is defined after the `+` sign.

A few `PROJ.4` parameters can be applied to most CRSs:

Parameter	Description
`+a`	Semimajor radius of the ellipsoid axis
`+axis`	Axis orientation
`+b`	Semiminor radius of the ellipsoid axis
`+ellps`	Ellipsoid name
`+k_0`	Scaling factor
`+lat_0`	Latitude of origin
`+lat_1 or 2`	Standard parallels
`+lon_0`	Central meridian
`+lon_wrap`	Center longitude to use for wrapping
`+over`	Allow longitude output outside -180 to 180 range, disables wrapping
`+pm`	Alternate prime meridian (typically a city name)
`+proj`	Projection name
`+units`	meters, US survey feet, etc.
`+vunits`	vertical units
`+x_0`	False easting
`+y_0`	False northing

Some parameters (not listed above) are specific to certain CRSs. Be sure to always verify the parameters that are allowed for each projection. You can't always "mix and match" the `PROJ.4` parameters when creating a custom projection.

These parameters can be confusing. To help with this we developed some visual examples.

Take for instance lambert conformal conic with the proj4 string of "+proj=lcc +lat_1=20 +lat_2=60 lon_0=-96 +datum=NAD83 +units=m". We can see that `lat_1` and `lat_2` specify the standard parallels. This is an example of a "secant" projection which touches the globe in two places in order to minimize distortion. The central meridian `lon_0` is moved away from Greenwich England 96 degrees W to be over roughly Dallas TX.

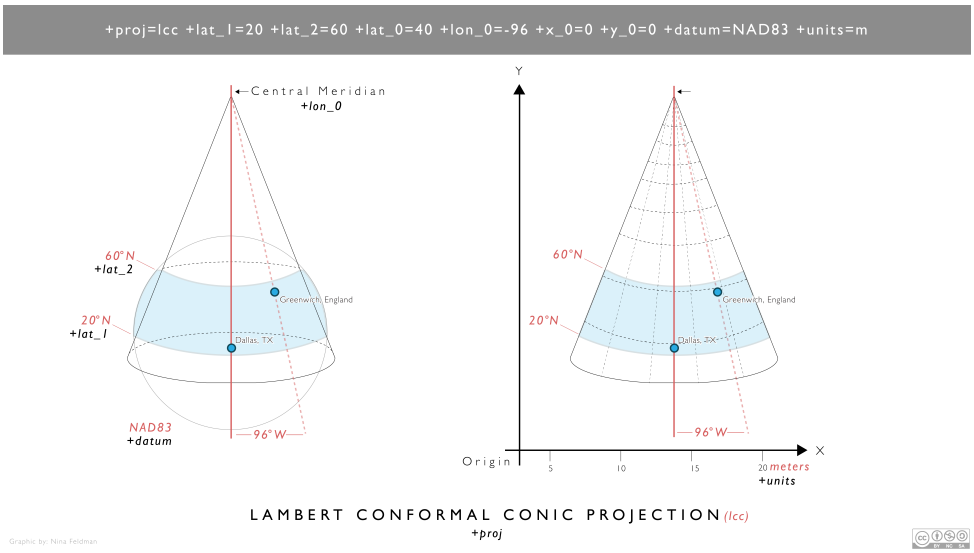


Fig. 16 Visualization of lambert conformal conic Proj4 string

We can look at another example for a UTM projection with the proj4 string of "+proj=tmerc +lon_0=-99 +x_0=500,000 units=m". This projection by comparison has a central meridian `lon_0` directly over Dallas TX which is 99 degrees W of Greenwich. This central meridan is assigned the 'false easting' `x_0` of 500,000 meters.

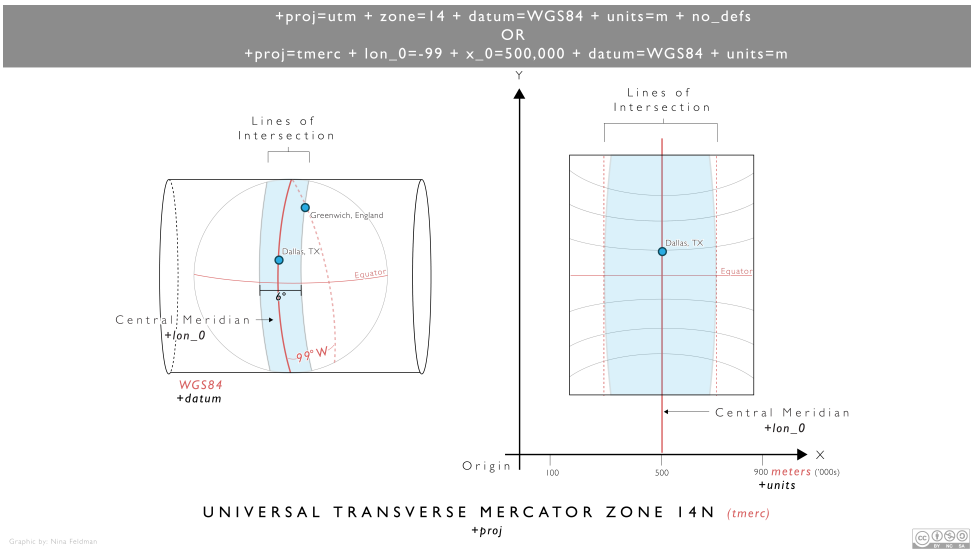


Fig. 17 Visualization of UTM zone 14 Proj4 string

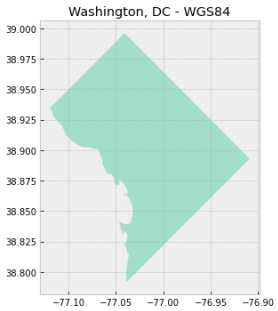
Sources: Cartographic projection; Using PROJ; Lesson 4. Understand EPSG, WKT and Other CRS Definition Styles, Leah Wasser

Creating a custom CRS using PROJ.4 string

We can change the parameters to create a custom CRS that suits our specific needs. In this section, we will create a CRS tailored to our DC shapefile.

Below is the original shapefile of DC that we read in above–this will serve as a reference point for comparison.

```
# Map original DC shapefile
map_data([dc], "Washington, DC - WGS84")
```



The map above utilizes a geographic coordinate system, as evidenced by the latitude and longitude values on the axes. Another way to check is to look at the geometry of the shapefile.

```
# Check geometry values of original shapefile
print("Geometry of shapefile:\n{}".format(dc['geometry'].head()))
```

```
Geometry of shapefile:
0    POLYGON ((-77.11980 38.93435, -77.11979 38.934...
Name: geometry, dtype: geometry
```

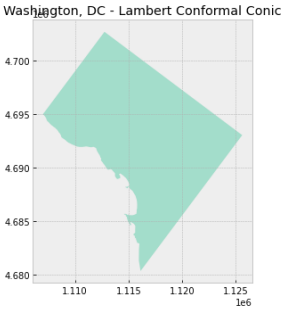
Looks like latitude and longitude coordinates!

Reprojecting shapefile to a projected coordinate system

Now, we will reproject the shapefile using a projected coordinate system. Here, we will use the Lambert Conformal Conic projection.

```
# Create a copy of the DC shapefile
dc_lcc = dc.copy()

# Reproject the data to Lambert Conformal Conic
# Source: https://proj.org/operations/projections/lcc.html
dc_lcc = dc_lcc.to_crs("+proj=lcc +lon_0=-90 +lat_1=33 +lat_2=45 +ellps=GRS80")

# Map reprojected DC shapefile
map_data([dc_lcc], "Washington, DC - Lambert Conformal Conic")

# Check geometry values
print("Geometry of shapefile:\n{}".format(dc_lcc['geometry'].head()))
```

```
Geometry of shapefile:
0    POLYGON ((1106986.814 4694966.592, 1106987.032...
Name: geometry, dtype: geometry
```



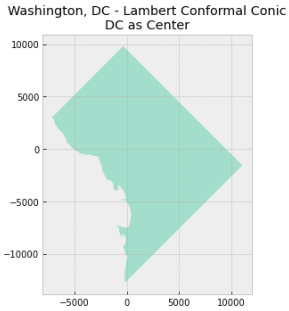
Notice how the values on the axes changed from latitude and longitude to meters.

Setting center point of projection

Also notice how the DC appears to be "tilted" to the left when compared to DC mapped using the previous CRS (WGS84). That's because the center of this projection is at the 90th meridian west (`+lon_0=-90`; negative is used to denote West) and the Equator (`+lat_0=0`; `0` is the default value when not specified in the `PROJ.4` string).

We can change the values for `lat_0` and `lon_0`, which refer to the latitude of origin and the central meridian, respectively. We will change those values to `38.9072` and `-77.0369`, which is the center of DC.

```
# Create a copy of the DC shapefile
dc_lcc_center = dc.copy()

# Reproject the data to Lambert Conformal Conic with specified center point
dc_lcc_center = dc_lcc_center.to_crs("+proj=lcc +lat_0=38.9072 +lon_0=-77.0369
+lat_1=33 +lat_2=45 +ellps=GRS80")

# Map reprojected DC shapefile
map_data([dc_lcc_center], "Washington, DC - Lambert Conformal Conic\nDC as
Center")
```



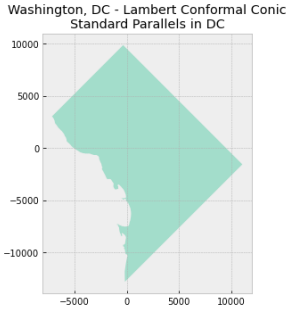
As seen in the new map above, the values on both axes have changed. The origin `(0, 0)` is now in the center of DC.

Setting and exploring standard parallels

The **`PROJ.4` string for this projection** has two additional parameters `lat_1` and `lat_2`, which specify the first and second parallel respectively. Recall that a conic projection "intersects" a globe at what is termed the standard parallels.

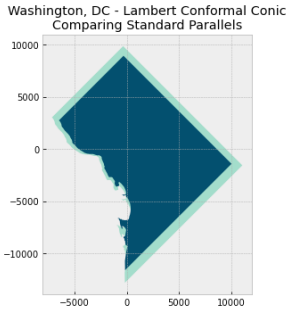
Here, we will set our two standard parallels at `38.850` and `39.950` as they fall within the DC boundaries.

```
# Create a copy of the DC shapefile
dc_lcc_parallels_1 = dc.copy()

# Reproject the data to Lambert Conformal Conic with specified center point and
standard parallels
dc_lcc_parallels_1 = dc_lcc_parallels_1.to_crs("+proj=lcc +lat_0=38.9072
+lon_0=-77.0369 +lat_1=38.850 +lat_2=39.950 +ellps=GRS80")

# Map reprojected DC shapefile
map_data([dc_lcc_parallels_1], "Washington, DC - Lambert Conformal Conic\nStandard
Parallels in DC")
```



Nothing really appears to be different; however, these values are important. Look what happens when we assign different values–ones that don't really make sense for mapping in DC–and compare the resulting DC map to the previous map.

```
# Create a copy of the DC shapefile
dc_lcc_parallels_2 = dc.copy()

# # Reproject the data to Lambert Conformal Conic with specified center point and
arbitrary standard parallels
dc_lcc_parallels_2 = dc_lcc_parallels_2.to_crs("+proj=lcc +lat_0=38.9072
+lon_0=-77.0369 +lat_1=10 +lat_2=60 +ellps=GRS80")

# Map reprojected DC shapefile
map_data([dc_lcc_parallels_1, dc_lcc_parallels_2], "Washington, DC - Lambert
Conformal Conic\nComparing Standard Parallels")
```



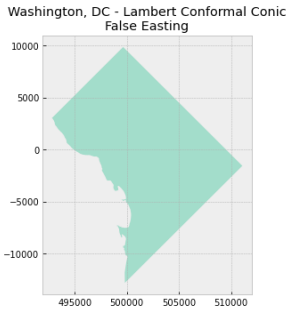
It appears DC has shrunk! Of course, that's not the case in reality, but because map projections are inherently imperfect, choosing arbitrary parameters can make things worse. In this case, since the chosen standard parallels are relatively far from each other and from DC, all the data in between the standard parallels end up being compressed even more. For more information, check out this Esri publication on map projections.

Setting false easting and false northing

Finally, two other parameters we can change are `+x_0` and `+y_0`, which are false easting and false northing respectively. The values assigned to these parameters simply offset the axes by the respective values; they do not change or affect the projection. False easting (with a value of 500,000 m) is used in Universal Tranverse Mercator (UTM) projections so that negative coordinates are avoided to the west of the central meridian in each zone. Similarly, false northing (with a value of 10,000,000 m) is also used to avoid negative coordinates when a UTM zone is in the southern hemisphere.

Let's set a false easting value of `500000` and a false northing value of `0`.

```
# Create a copy of the DC shapefile
dc_lcc_false_e = dc.copy()

# Reproject the data to Lambert Conformal Conic with specified center point,
standard parallels, and false easting/northing
dc_lcc_false_e = dc_lcc_false_e.to_crs("+proj=lcc +lat_0=38.9072 +lon_0=-77.0369
+lat_1=38.850 +lat_2=39.950 +x_0=500000 +y_0=0 +ellps=GRS80")

# Map reprojected DC shapefile
map_data([dc_lcc_false_e], "Washington, DC - Lambert Conformal Conic\nFalse
Easting")
```



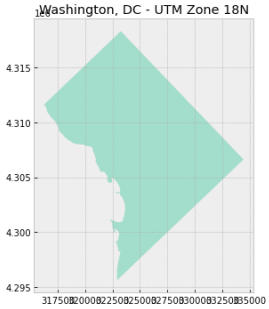
Notice that the x-axis shifted to the left by 500,000 meters. The y-axis stayed the same because we did not use false northing. Now, the "origin" is `(500000, 0)`.

Source: Universal Transverse Mercator system, Hunter College Department of Geography and Environmental Science

Comparing custom projection to Universal Tranverse Mercator

UTM is conformal. So let's compare our custom projection to UTM Zone 18N (in which DC falls).

```
# Create a copy of the DC shapefile
dc_utm18n = dc.copy()

# Reproject the data to UTM Zone 18N
# Source: https://www.spatialreference.org/ref/epsg/wgs-84-utm-zone-18n/
dc_utm18n = dc_utm18n.to_crs("+proj=utm +zone=18 +ellps=WGS84 +datum=WGS84
+units=m +no_defs")

# Map reprojected DC shapefile
map_data([dc_utm18n], "Washington, DC - UTM Zone 18N")
```



Looks pretty much the same as our custom one!

Source: Universal Transverse Mercator, Esri

EPSG Codes

Many CRSs are assigned and can be referenced by an `EPSG` code, which consists of a four or five digit number. `EPSG` stands for the European Petroleum Survey Group, a now-defunct organization that compiled this CRS dataset. `EPSG` codes can be further explored with the EPSG Geodetic Parameter Dataset or at SpatialReference.org.

Sources: Overview of Coordinate Reference Systems (CRS) in R, University of California, Santa Barbara; Lesson 4. Understand EPSG, WKT and Other CRS Definition Styles, Leah Wasser

Obtaining EPSG code from PROJ.4 string

Not all CRSs have a corresponding `EPSG` code, but we can find the `EPSG` code (if it exists) given a `PROJ.4` string. To do so, we can use the `to_espg()` function in the `pyproj` module.

```
# Import module
import pyproj

def get_epsg(proj4_string, min_confidence = 70):
    '''Function takes a PROJ.4 string and optional minimum confidence level as
inputs and outputs the relevant EPSG code, if one exists. Source:
https://geopandas.org/docs/user_guide/projections.html'''

    # Get relevant EPSG at the specified minimum confidence level
    return pyproj.CRS(proj4_string).to_epsg(min_confidence = min_confidence)


# Set variable to the PROJ.4 string of NAD83 / California zone 3
# Source: https://www.spatialreference.org/ref/epsg/26943/
proj4_full = "+proj=lcc +lat_1=38.43333333333333 +lat_2=37.06666666666667
+lat_0=36.5 +lon_0=-120.5 +x_0=2000000 +y_0=500000 +ellps=GRS80 +datum=NAD83
+units=m +no_defs"

# Call function to obtain relevant EPSG code
result = get_epsg(proj4_full)

# Print result
print("EPSG code for {}: {}.".format(proj4_full, result))
```

```
EPSG code for +proj=lcc +lat_1=38.43333333333333 +lat_2=37.06666666666667
+lat_0=36.5 +lon_0=-120.5 +x_0=2000000 +y_0=500000 +ellps=GRS80 +datum=NAD83
+units=m +no_defs: 26943.
```

Success! In the example above, we were able to successfully obtain the `EPSG` code (in this case, `26943` for NAD83 / California zone 3) because it is exactly matched to the `PROJ.4` string. If we're missing some information in the `PROJ.4` string, however, we might not be able to get an exact `EPSG` code match.

```
# Set variable to the PROJ.4 string of NAD83 / California zone 3, with a few
parameters missing
proj4_missing = "+proj=lcc +lat_1=38.43333333333333 +lat_2=37.06666666666667
+lat_0=36.5 +lon_0=-120.5 +ellps=GRS80 +datum=NAD83 +units=m +no_defs"

# Call function to obtain relevant EPSG code
result_missing = get_epsg(proj4_missing)

# Print result
print("EPSG code for {}: {}.".format(proj4_missing, result_missing))
```

```
EPSG code for +proj=lcc +lat_1=38.43333333333333 +lat_2=37.06666666666667
+lat_0=36.5 +lon_0=-120.5 +ellps=GRS80 +datum=NAD83 +units=m +no_defs: None.
```

Source: Managing Projections: Coordinate Reference Systems, GeoPandas

Lowering minimum confidence parameter

We can lower the min_confidence parameter value in the `to_epsg()` function, which will cause the function to return an `EPSG` code that is the closest match to the provided `PROJ.4` string.

```
# Call function to obtain relevant EPSG code and lower the minimum confidence
value
result_lower_confidence = get_epsg(proj4_missing, min_confidence = 20)

# Print result
print("EPSG code for {}: {}.".format(proj4_missing, result_lower_confidence))
```

```
EPSG code for +proj=lcc +lat_1=38.43333333333333 +lat_2=37.06666666666667
+lat_0=36.5 +lon_0=-120.5 +ellps=GRS80 +datum=NAD83 +units=m +no_defs: 26943.
```

Source: Managing Projections: Coordinate Reference Systems, GeoPandas

To get the `PROJ.4` string from an `EPSG` code, we can use the **`rasterio`** module.

Sources: Overview of Coordinate Reference Systems (CRS) in R, University of California, Santa Barbara; Chapter 9: Coordinate Systems, Manuel Gimond; Coordinate Systems: What's the Difference?, Esri; Map projections, Henrikki Tenkanen; Map projections, Henrikki Tenkanen & Vuokko Heikinheimo

Learning Objectives

- Learn how to use Affine transforms
- Differentiate translate, rotate, skew, shear

Review

- Data Structures
- Vector Data
- Raster Data
- Matrix Algebra Intro

Affine Transforms

Affine transformations allows us to use simple systems of linear equations to manipulate any point or set of points. It allows us to move, stretch, or even rotate a point or set of points. In the case of GIS, it is used to distort raster data, for instance satellite imagery, to fit a new projection or CRS.

Fig. 18 Example of a warped (reprojected) image

First some general properties of affine transforms:

- **Preserves**
  - Points, straight linear & planes
  - Sets of parallel lines
  - Ratio of distances between points on same straight line
- **Distorts**
  - Angle between lines
  - Distance between points

Types of Transformations

There are four core ways that you can manipulate an image. These are called “Transforms”:

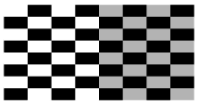
Transform	Description	Example
Translation	Moves a set of points some fixed distance in the x and y plane	
Scale	Increases or decreases the scale, or distance between points in the x and y plane	
Rotate	Rotates points around the origin, or some defined axis	
Shear	Shifts points in proportion to any given points x and y coordinate	

Image Credit: Wikipedia

In combination we can warp any point, or set of points (e.g. raster image) into a new projection. This is the equivalent of reprojection for vector data. In order to implement these transforms we will need to learn about the math behind *Affine Transforms*! Yeah!

Simple Transform Examples

To simplify things first lets think about how to do transformations of a single point in a 2d space.

For any location $\mathbf{x} = (x, y)$ we can transform $\mathbf{x}$ to $\dot{\mathbf{x}}$ using simple linear adjustments. Here we can think of point $\mathbf{x}$ as being stored as an array:

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} \text{can be transformed to } \acute{\mathbf{x}} = \begin{bmatrix} \acute{x} \\ \acute{y} \end{bmatrix}$$

From here we can transform the values in the x and y position using scaler values a through f:

$$\acute{\mathbf{x}} = \begin{bmatrix} ax + by + c \\ dx + ey + f \end{bmatrix}$$

> **Note**
>
> Looking at the formula above, we are adjusting the value of x with $ax + by + c$, so x can be multiplied (scaled) by some value a, it can also be scaled based on the y value with $+by$, or simply adjusted up or down by the value $+c$

To understand how this works, let's walk through the basic transformations of $\mathbf{x}$:

If we multiply x and y by one, by setting $a, e = 1$, and zero out the effect of the other axis, by setting $b, d = 0$, we have a simple case of translation, where x moves right by c and y up by the value of f:

$$\acute{\mathbf{x}} = \begin{bmatrix} 1x + 0y + c \\ 0x + 1y + f \end{bmatrix}$$

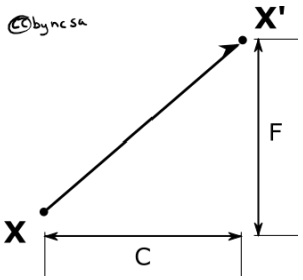


Fig. 19 Translate a coordinate

We can scale x and y by setting a and e to 0.5 (or some fraction), and setting all other values to zero ($b, d, c, f = 0$):

$$\acute{\mathbf{x}} = \begin{bmatrix} ax \\ ey \end{bmatrix}$$

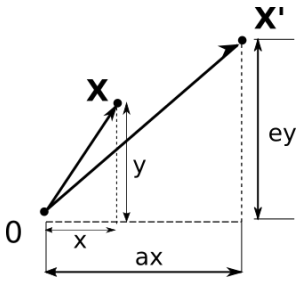


Fig. 20 Scale a coordinate

We can rotate a point around the origin by setting $a, e = \cos\theta$, $b = -\sin\theta$ and $c, f = 0$:

$$\acute{\mathbf{x}} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

where θ is the angle of rotation (counterclockwise) around the origin.

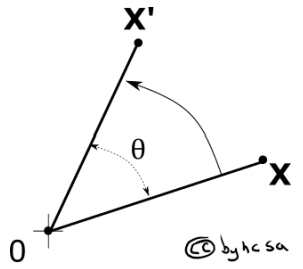


Fig. 21 Rotate a coordinate

Finally by adjusting x based on the value of y (and visa versa), we can achieve a shear transform:

$$\acute{\mathbf{x}} = \begin{bmatrix} x + by \\ y + dx \end{bmatrix}$$

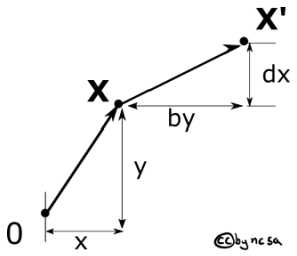


Fig. 22 Shear transform a coordinate

Transforming Matrices

It is often convenient to represent these equations as matrices. This allows us to easily chain together a series of operations. We can represent our transformed point $\acute{\mathbf{x}}$ as follows:

$$\begin{bmatrix} \acute{x} \\ \acute{y} \end{bmatrix} = \begin{bmatrix} ax + by \\ dx + ey \end{bmatrix} = \begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

> **Note**
>
> If you aren't familiar with how matrix multiplication works please watch this.

In this new context we can easily do a scale, rotate or shear transform by replacing the matrix of a, b, d, e with:

$$\text{Rotate: } \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$\text{Scale: } \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

$$\text{Shear: } \begin{bmatrix} 1 & r_x \\ r_y & 1 \end{bmatrix}$$

But wait, what about the easiest transform, "translation"? Unfortunately that makes things a little more complicated! But not that complicated.

In order to be able to perform a translate in matrix form we need to extend our matrices, adding one row along the bottom. In the following form we can now perform all the basic transformations to calculate $\acute{\mathbf{x}}$:

$$\begin{bmatrix} \acute{x} \\ \acute{y} \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}$$

Now that we have scalers c and f all the transforms are possible. We do however, need to update our previous operations:

$$\text{Translate: } \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

Where Δx shifts in the x axis and Δy determines the shift in the y axis.

$$\text{Rotate: } \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Scale: } \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Shear: } \begin{bmatrix} 1 & r_x & 0 \\ r_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Numeric Examples

Translate

Now let's assume we have a point a at (-2,-2) for (x,y). For simplicity sake lets assume we want to move it up to the origin by adding 2 to both x and y.

Let's start by defining our point in our matrix form:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ -2 \\ 1 \end{bmatrix}$$

Let's get our transform matrix M to perform our translate, where $\Delta x, \Delta y = 2$ because we want to move it up and to the right:

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

We can then multiply the two:

$$\begin{bmatrix} -2 \\ -2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -2 \times 1 + -2 \times 0 + 1 \times 2 \\ -2 \times 0 + -2 \times 1 + 1 \times 2 \\ -2 \times 0 + -2 \times 0 + 1 \times 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Congrats you reached (0,0), just like you always dreamed!

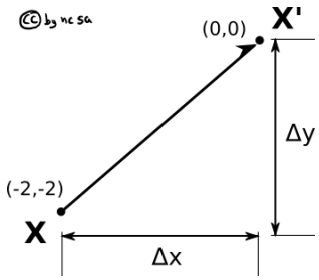


Fig. 23 Moving a point

> **Note**
>
> Remember the bottom row can be ignored because $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

Rotate

All the transformations follow the same procedure, let's try rotation just to make sure that we have it figured out. Let's rotate our point at (-2,-2) by 180 degrees around the origin:

$$\begin{bmatrix} -2 \\ -2 \\ 1 \end{bmatrix} \begin{bmatrix} \cos 180 & -\sin 180 & 0 \\ \sin 180 & \cos 180 & 0 \\ 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} -2 \times \cos 180 + -2 \times -\sin 180 + 1 \times 0 \\ -2 \times \sin 180 + -2 \times \cos 180 + 1 \times 0 \\ 1 \times 0 + 1 \times 0 + 1 \times 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}$$

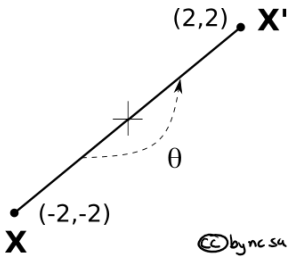


Fig. 24 Rotate a point

> **Learning Objectives**
>
> - How to assign a projection to vector data
> - How to reproject vector data

> **Review**
>
> - Understanding CRS codes
> - Creating Points, Lines, Polygons

Vector Coordinate Reference Systems (CRS)

When it comes to coordinate reference systems points, lines and polygons are convenient because each point or node has an assigned coordinate pair (x,y). The only trick then is to know how those coordinates relate to the coordinate space, or location on the ground.

Define a Projection for Points, Lines, Polygons

When a point, line or polyon is created each point or node has two coordinates `x` and `y`. The location of these two points on the ground will change wildly between projections. The coordinate pair (0,0) might mean a location just off shore from Ghana with WGS84 LatLon, or in the middle of the Pacific ocean in another.

Let's take a look at the example of a polygon with coordinates (0,45),(5,45),(5,40),(0,40) below. In the left pane we can see that, although we have the polygon's node coordinates, we don't know where they are located! This is because no coordinate reference system has been assigned to it. Could be in outer space for all we know. On the right, we can see that we have assigned it WGS84 geographic lat lon i.e. `EPSG:4326`. Suddenly the coordinates make sense, because we know how they relate to locations on the ground.

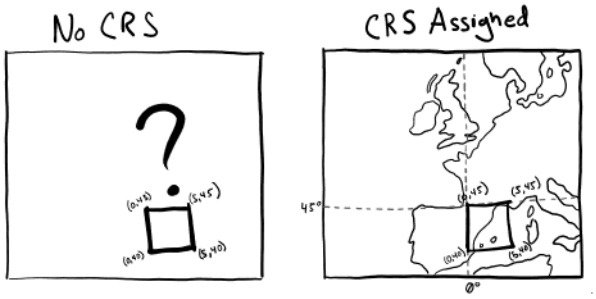
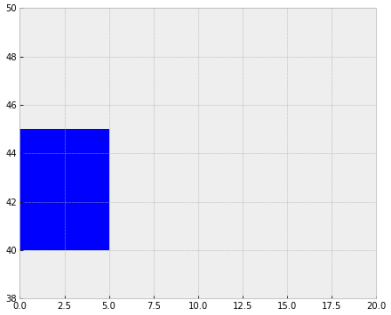


Fig. 25 Example of assigning a coordinate reference system

Every time we create vector data (or receive it from someone else), we need to make sure that a projection is assigned to it.

The following demonstrates how to

```
import geopandas as gpd
from shapely.geometry import Point, Polygon
import matplotlib.pyplot as plt
plt.style.use('bmh')

a_square = {'name': ['Washington\n(38.90, -77.03)' ],
     'geometry': [Polygon([Point(0,45), Point(5,45), Point(5,40), Point(0,40)])]}

# create a dataframe from the nodes and assign the CRS
gdf_square = gpd.GeoDataFrame(a_square, crs="EPSG:4326")
fig, ax = plt.subplots(figsize=(12, 6))
gdf_square.plot(ax=ax)
plt.ylim(38, 50)
plt.xlim(0, 20)
plt.show()
```



> **Note**
>
> Take a moment to review proj4 string and EPSG codes here

Reproject Points, Lines, Polygons

Once a projection is assigned we often have to 'reproject' it to another one. Reprojection entails using a set of formulas to convert (x,y) stored in latitude and longitude into another coordinate space. This entails a two step process.

Looking at the example below, we will move from Sinosoidal to Hobo-Dyer. The first step however is to use the "inverse equation" to convert coordinate pairs from Sinosoidal back to lat lon, and the use the forward equation to convert lat lon into the Hobo-Dyer coordinate space.

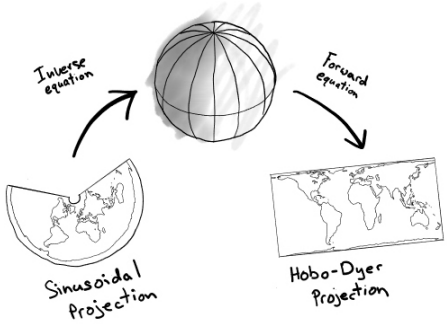


Fig. 26 Reprojecting vectors with inverse and forward equations

These 'forward' and 'inverse' equations can be simple or complex. To make things easy to understand lets look at the example of reprojecting from proj4 code `+proj=longlat +datum=WGS84` to `+proj=longlat +datum=WGS84 +lon_0=-10`, where `+lon_0=-10` just moves the prime meridian 10 degrees west (west is negative). Because we are already in latitude and longitude we can ignore the 'inverse' equation and just look at the 'forward' equation.

In this case the 'forward' equation is very simple:

$$x = x + 10$$

$$y = y$$

As result the polygon remains in the same location, but in the 'new' projection longitude values are now all 10 degrees higher.

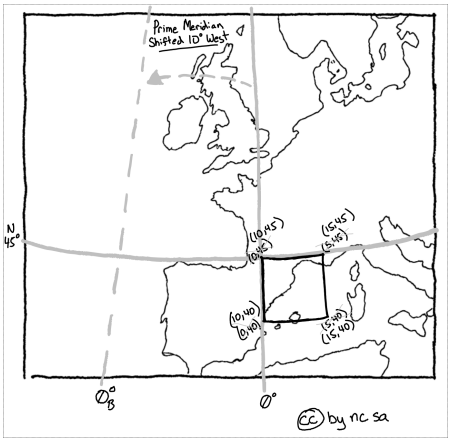


Fig. 27 Reprojecting by moving the prime meridian west

Most 'forward' and 'inverse' equations are non-linear and more complex. Take for instance the Gall-Peters forward projection equations are shown below:

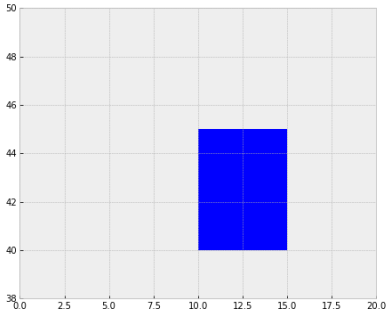
$$x = \frac{R\pi\lambda\cos 45^\circ}{180^\circ}$$

$$y = \frac{R\sin\varphi}{\cos 45^\circ}$$

where λ is the longitude from the central meridian in degrees, φ is the latitude, and R is the radius of the globe used as the model of the earth for projection. Luckily computers make all these calculation quick and easy.

Refering back to our previous example, let's use geopandas to move the prime meridian 10 degrees west:

```
# reproject the data
gdf_square_10w = gdf_square.to_crs("+proj=longlat +datum=WGS84 +lon_0=-10")

fig, ax = plt.subplots(figsize=(12, 6))
gdf_square_10w.plot(ax=ax)
plt.ylim(38, 50)
plt.xlim(0, 20)
plt.show()
```



Note the shift in coordinates along x by 10 degrees west.

Learning Objectives

- Learn how rasters are reprojected
- Learn how affine transforms are used
- Use rasterio to reproject a raster
- Learn about interpolation options during transforms

Review

- Intro to Raster data
- Affine transformation

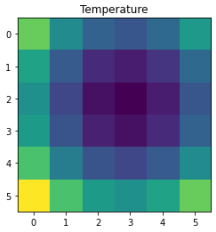
Raster Coordinate Reference Systems (CRS)

Raster data is very different that vector data, one of the key differences is that we don't have a pair of coordinates (x,y) for each pixel in a raster. How then do we know where the raster is located in addition to what the data values are? For a new spatial raster (e.g. geotif) we need to store a few other pieces of information seperately. We need to keep track of the location of the upper left hand corner, the resolution (in both the x and y direction) and a description of the coordinate space (i.e. the CRS), amongst others.

Fig. 28 Example of a warped (reprojected) image

Let's start from the `ndarray` `Z` that we want to span from [-90°,90°] longitude, and [-90°,90°] latitude. For more detail on the construction of these arrays please refer to the raster section.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-90, 90, 6)
y = np.linspace(90, -90, 6)
X, Y = np.meshgrid(x, y)

Z1 =  np.abs(((X - 10) ** 2 + (Y - 10) ** 2) / 1 ** 2)
Z2 =  np.abs(((X + 10) ** 2 + (Y + 10) ** 2) / 2.5 ** 2)
Z =  (Z1 - Z2)

plt.imshow(Z)
plt.title("Temperature")
plt.show()
```



Note that *`Z` contains no data on its location*. Its just an array, the information stored in `x` and `y` aren't associated with it at all. This location data will often be stored in the header of file. In order to 'locate' the array on the map we will use affine transformations.

Affine transformations allows us to use simple systems of linear equations to manipulate any point or set of points (review affine transforms here). It allows us to move, stretch, or even rotate a point or set of points. In the case of GIS, it is used to move raster data, a satellite image, to the correct location in the CRS coordinate space.

Describing the Array Location (Define a Projection)

In this example the coordinate reference system will be '+proj=latlong', which describes an equirectangular coordinate reference system with units of decimal degrees. Although `X` and `Y` seems relevant to understanding the location of cell values, `rasterio` instead uses affine transformations instead. Affine transforms uses matrix algebra to describe where a cell is located (translation) and what its resolution is (scale). Review affine transformations and see an example here.

The affine transformation matrix can be computed from the matrix product of a translation (moving N,S,E,W) and a scaling (resolution). First, we start with translation where Δx and Δy define the location of the upper left hand corner of our new `Z` ndarray. As a reminder the translation matrix takes the form:

$$\text{Translate} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

Now we can define our translation matrix using the point coordinates (`x[0]`,`y[0]`), but these need to be offset by 1/2 the resolution so that the cell is centered over the coordinate (-90,90). Notice however there are some difference between the x and y resolution:

Both arrays have the same spatial resolution

```
xres = (x[-1] - x[0]) / len(x)
xres
```

```
30.0
```

But notice that the y resolution is **negative**:

```
yres = (y[-1] - y[0]) / len(y)
yres
```

```
-30.0
```

We need to create our translation matrix by defining the location of the upper left hand corner. Looking back at our definitions of our coordinates `X` and `Y` we can see that they are defined with `x = np.linspace(-90, 90, 6); y = np.linspace(90, -90, 6); X, Y = np.meshgrid(x, y)`, if you run `print(X);print(Y)` you will see that the **center** of the upper left hand corner should be located at (-90,90). The **upper left hand corner** of that same cell is actually further up and to the left than (-90,90). It follows then that that corner should be shifted exactly 1/2 the resolution of the cell, both up and to the left.

We can therefore define the upper left hand corner by setting $\Delta x = x[0] - xres/2$ and $\Delta y = y[0] - yres/2$. Remember yres is negative, so subtraction is correct.

```
from rasterio.transform import Affine
print(Affine.translation(x[0] - xres / 2, y[0] - yres / 2))
```

```
| 1.00, 0.00,-105.00|
| 0.00, 1.00, 105.00|
| 0.00, 0.00, 1.00|
```

We also need to scale our data based on the resolution of each cell, the scale matrix takes the following form:

$$\text{Scale} = \begin{bmatrix} xres & 0 & 0 \\ 0 & yres & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
print(Affine.scale(xres, yres))
```

```
| 30.00, 0.00, 0.00|
| 0.00,-30.00, 0.00|
| 0.00, 0.00, 1.00|
```

We can do both operations simultaneously in a new `transform` matrix by calculating the product of the Translate and Scale matrix:

$$\text{Translate} \cdot \text{Scale} = \begin{bmatrix} xres & 0 & \Delta x \\ 0 & yres & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

```
transform = Affine.translation(x[0] - xres / 2, y[0] - yres / 2) *
Affine.scale(xres, yres)
print(transform)
```

```
| 30.00, 0.00,-105.00|
| 0.00,-30.00, 105.00|
| 0.00, 0.00, 1.00|
```

Now we need to write out a `tif` file that holds the data in `Z` and its data type with `dtype`, the location described by `transform`, in coordinates described by the coordinate reference system `+proj=latlon`, the number of 'bands' of data in `count` (in this case just one), and the shape in `height` and `width`.

```
import rasterio
with rasterio.open(
    '../temp/Z.tif',
    'w',
    driver='GTiff',
    height=Z.shape[0],
    width=Z.shape[1],
    count=1,
    dtype=Z.dtype,
    crs='+proj=latlong',
    transform=transform,
) as dst:
    dst.write(Z, 1)
```

All this info is stored in `dst` and then written to disk with `dst.write(Z,1)`. Where `write` gets the array of data `Z` and the band location to write to, in this case band `1`. This is a bit awkward, but I believe is a carryover from GDAL which rasterio relies on heavily (like all other platforms including arcmap etc).

The Crazy Tale of the Upper Left Hand Corner

To help us understand what is going on with `transform` it helps to work an example. For our example above we need to define the translate matrix that helps define the upper left hand corner of our rainfall raster data `Z`. In particular we need the upper left cell center to be located at (-90,90), so the upper left hand corner need to be 1/2 the resolution above and to the left of (-90,90), implying a location of (-105,105) since the resolution is 25 degrees.

We can visualize what we need to do here:

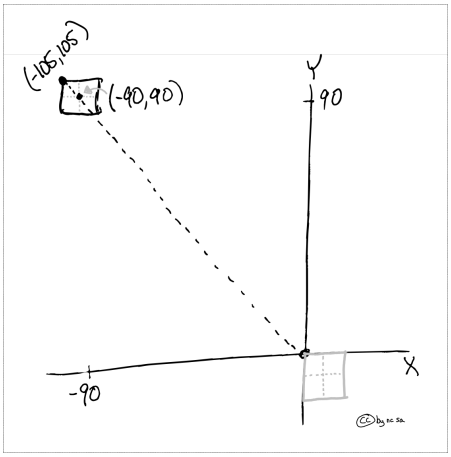


Fig. 29 Example of using translation to shift the upper left hand corner

Let's walk through the math behind the scenes. Here we use our transform matrix to move our upper left hand corner which is assumed to start at the origin (0,0).

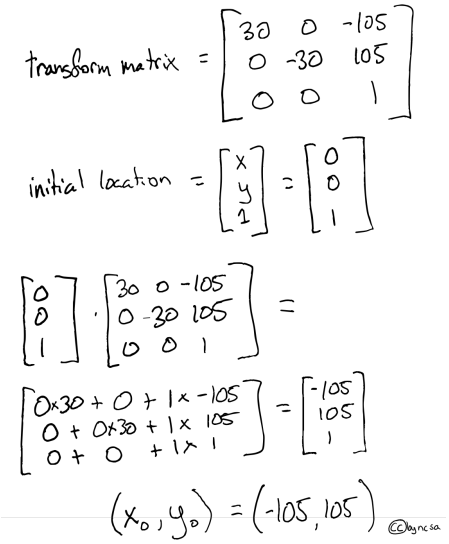


Fig. 30 Example of using affine translation of a matrix to shift the upper left hand corner

The final coordinate of the upper left hand corner are $(x_0, y_0) = (-105, 105)$

Translate is a "map"

Now here's the magic, our new `translate` matrix can be used to easily find the coordinates of any cell based on its row and column number. To see how if works, we are going to multiply our `translate` matrix by `(column_number, row_number)` to retrieve the coordinates of that cell's upper right hand corner. Essentially, `translate` "maps" row and column indexes to coordinates! OMG! This is fun… ok kidding, but it's useful.

Let's see how we can calculate a few coordinates (upper left) based on the visual examples below:

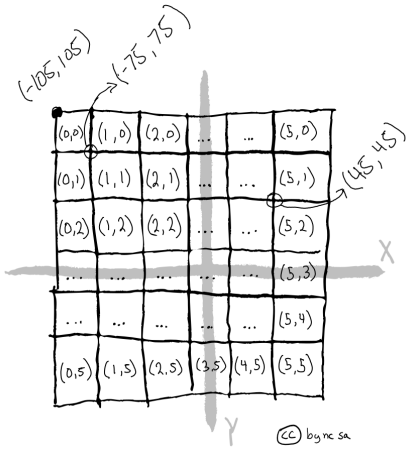


Fig. 31 Example of using transform to identify coordinates based on row and column

Let's start with the easiest and retrieve the upper left corner coordinates based on `transform * (row_number, column_number)`:

```
print(transform*(0,0))
```

```
(-105.0, 105.0)
```

Let's find the corner that is one cell down (-30°) and to the right (+30°)

```
print(transform*(1,1))
```

```
(-75.0, 75.0)
```

Just to make sure it works let's find a harder one, 5th column right, 2nd row down:

```
print(transform*(5,2))
```

```
(45.0, 45.0)
```

How Transforms Works

Let's work the example of finding the upper left coordinates of with `row=5, column=2`:

$$\begin{bmatrix}5\\2\\1\end{bmatrix}\begin{bmatrix}30 & 0 & -105\\0 & -30 & 105\\0 & 0 & 1\end{bmatrix} = \begin{bmatrix}5\times 30+2\times 0+1\times -105\\5\times 0+2\times -30+1\times 105\\5\times 0+2\times 0+1\times 1\end{bmatrix} = \begin{bmatrix}45\\45\\1\end{bmatrix}$$

Wow, it works! Come on it's at least a little bit cool. Depending on your definition of cool.

Reproject a Raster - The Simple Case

How then do we reproject a raster? Since `transform` is a map of pixel locations, warping a raster then becomes as simple as knowing the `transform` of your destination based on the description of the new coordinate reference system (CRS). If you haven't please study affine transformations.

Shifting the Prime Meridian

One of the easiest cases is that of false easting, or moving the prime meridian. Let's walk through an example where we start with a raster with an upper left hand corner at (0, 45), then we will apply a transform to move it to (10, 45) by moving the prime meridian 10° to the west (e.g. using `+lon_0=-10` from the proj4string).

Let's start be looking visually at what we plan to do:

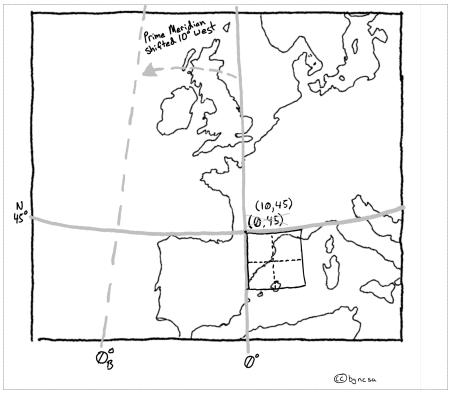


Fig. 32 Example of using translate to reproject an image by moving prime meridian

We can then use our knowledge of matrix algebra and transform matrices to solve for the new upper left hand corner coordinate (x_B, y_B)

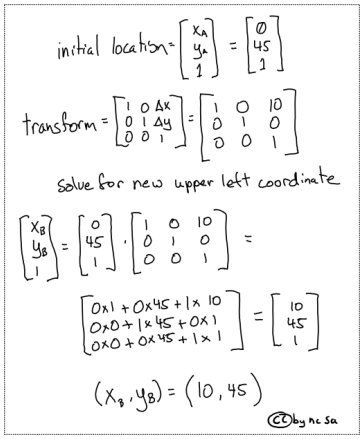


Fig. 33 Example of using translate matrix to reproject an image by moving prime meridian

Reproject a Raster - The Complex Case

In many cases reprojecting a raster requires changing the number of rows or columns, or 'warping' (i.e. bending) an image. All of these examples create a problem, the centroids of the new projected raster don't line up with the centroids of the original raster. Therefore they now represent locations on the ground that weren't in the original dataset.

Take for instance the case of a 'warped' raster image (below) which for instance occurs when you switch from a spherical CRS (like lat lon) to a projected (or flat) CRS. Notice that the centroids of the two rasters no longer over lap:

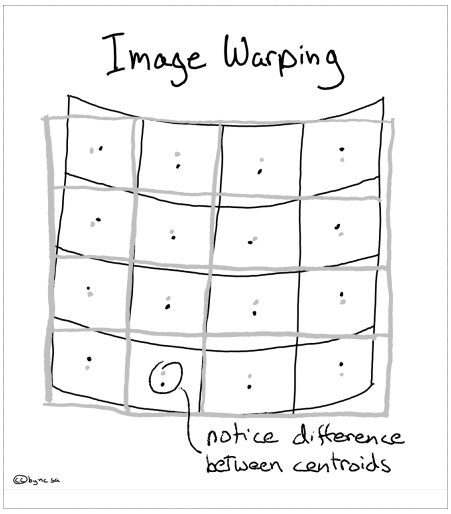


Fig. 34 Example of warping an image during reprojection

In this case we have a decision to make, how will we assign values to the new warped raster? Keep in mind the values must change because they now point to different locations on the ground. For this we have a number of 'interpolation' options, some simple, some complex.

> **Note**
>
> What is interpolation? Interpolation allows us to make an informed guess of a value at a new location. Read more here

Interpolation Options

There are three commonly used interpolation methods: a) Nearest neighbor - assigns the value of the nearest centroid, b) bilinear interpolation - uses a straight line between know locations, c) bicubic interpolation - uses curved line between known locations.

In the visual example below we will try to estimate the value for location `C` based on the known values at locations `A` and `B`.

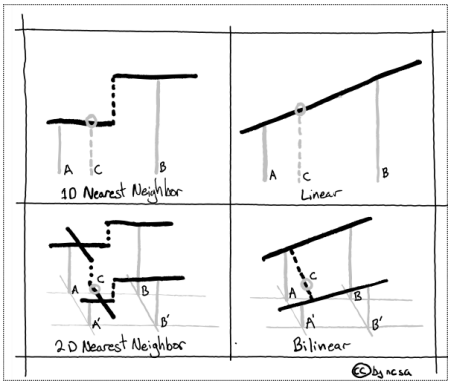


Fig. 35 Example of nearest neighbor and bilinear interpolation

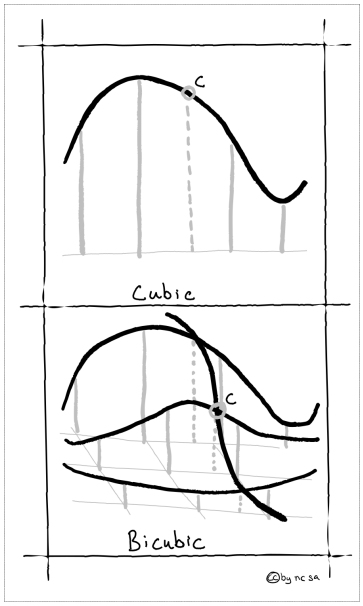


Fig. 36 Example of nearest neighbor and bicubic interpolation

Choosing the Right Interpolation Method

Choosing the correct interpolation method is important. The following table should help you to decided. Remember categorical data might include land cover classes (forest, water, etc), and continuous data is measurable for instance rainfall (values 0 to 20mm).

Method	Description	Fast	Categorical	Continuous
Nearest Neighbor	Assigns nearest value	Y	Y	
Bilinear	Linear estimation	Y		Y
Bicubic	Non-Linear estimation	Most of the time		Y

For categorical data, Nearest Neighbor is your only choice, enjoy it. For continuous data, like quantity of rain, you can choose between Bilinear and Bicubic (i.e. “cubic convolution”). For most data Bilinear interpolation is fast and effective. However if you believe your data is highly non-linear, or widely spaced, you might consider using Bicubic. Some experimentation here is often informative.

Reprojecting a Raster with Rasterio

Ok enough talk already, how do we reproject a raster? Before we get into it, we need to talk some more… about `calculate_default_transform`. `calculate_default_transform` allows us to generate the transform matrix required for the new reprojected raster based on the characteristics of the original and the desired output CRS. Note that the `source` (src) is the original input raster, and the `destination` (dst) is the outputed reprojected raster.

First, remember that the transform matrix takes the following form:

$$\text{Transform} = \begin{bmatrix} xres & 0 & \Delta x \\ 0 & yres & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

Now let’s calculate the tranform matrix for the destination raster:

```
import numpy as np
import rasterio
from rasterio.warp import reproject, Resampling, calculate_default_transform

dst_crs = "EPSG:3857"  # web mercator(ie google maps)

with rasterio.open("../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF") as src:

    # transform for input raster
    src_transform = src.transform

    # calculate the transform matrix for the output
    dst_transform, width, height = calculate_default_transform(
        src.crs,    # source CRS
        dst_crs,    # destination CRS
        src.width,    # column count
        src.height,  # row count
        *src.bounds,  # unpacks outer boundaries (left, bottom, right, top)
    )

print("Source Transform:\n",src_transform,'\n')
print("Destination Transform:\n", dst_transform)
```

```
Source Transform:
 | 30.00, 0.00, 717345.00|
| 0.00,-30.00,-2776995.00|
| 0.00, 0.00, 1.00|

Destination Transform:
 | 33.24, 0.00,-6105300.09|
| 0.00,-33.24,-2885952.71|
| 0.00, 0.00, 1.00|
```

Notice that in order to keep the same number of rows and columns that the resolution of the destination raster increased from 30 meters to 33.24 meters. Also the coordinates of the upper left hand corner have shifted (check $\Delta x, \Delta x$).

Ok finally!

```
dst_crs = "EPSG:3857"  # web mercator(ie google maps)

with rasterio.open("../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF") as src:
    src_transform = src.transform

    # calculate the transform matrix for the output
    dst_transform, width, height = calculate_default_transform(
        src.crs,
        dst_crs,
        src.width,
        src.height,
        *src.bounds,  # unpacks outer boundaries (left, bottom, right, top)
    )

    # set properties for output
    dst_kwargs = src.meta.copy()
    dst_kwargs.update(
        {
            "crs": dst_crs,
            "transform": dst_transform,
            "width": width,
            "height": height,
            "nodata": 0,  # replace 0 with np.nan
        }
    )

    with rasterio.open("../temp/LC08_20200518_webMC.tif", "w", **dst_kwargs) as
dst:
        for i in range(1, src.count + 1):
            reproject(
                source=rasterio.band(src, i),
                destination=rasterio.band(dst, i),
                src_transform=src.transform,
                src_crs=src.crs,
                dst_transform=dst_transform,
                dst_crs=dst_crs,
                resampling=Resampling.nearest,
            )
```

Fig. 37 Reprojected Landsat Image

Source: Creating Raster Data

Learning Objectives

- Create and manipulate vector attributes
- Subset data
- Plot lat lon as points
- Subset points by location

Review

- Vector Data

Attributes & Indexing for Vector Data

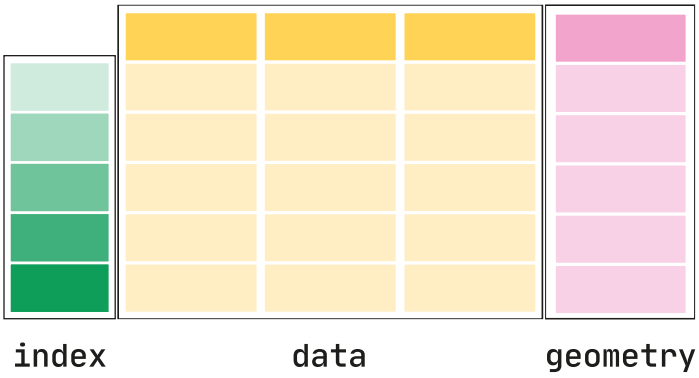


Fig. 38 Structure of a `GeoDataFrame` extends the functionality of a Pandas `DataFrame`

Each `GeoSeries` can contain any geometry type (e.g. points, lines, polygon) and has a `GeoSeries.crs` attribute, which stores information on the projection (CRS stands for Coordinate Reference System). Therefore, each `GeoSeries` in a `GeoDataFrame` can be in a different projection, allowing you to have, for example, multiple versions of the same geometry, just in a different CRS.

Tip

Becuase GeoPandas are so intertwined spend the time to learn more about here Pandas User Guide

Create New Attributes

One of the most basic operations is creating new attributes. Let's say for instance we want to look at the world population in millions. We can start with an existing column of data `pop_est`. Let's start by looking at the column names:

```
import geopandas
world = geopandas.read_file(geopandas.datasets.get_path('naturalearth_lowres'))
world.columns
```

```
Index(['pop_est', 'continent', 'name', 'iso_a3', 'gdp_md_est', 'geometry'],
dtype='object')
```

We can then do basic operations on the basis of column names. Here we create a new column `m_pop_est`:

```
world['m_pop_est'] = world['pop_est'] / 1e6
world.head(2)
```

	pop_est	continent	name	iso_a3	gdp_md_est	geometry	m_pop_est
0	920938	Oceania	Fiji	FJI	8374.0	MULTIPOLYGON (((180.00000 -16.06713, 180.00000...	0.920938
1	53950935	Africa	Tanzania	TZA	150600.0	POLYGON ((33.90371 -0.95000, 34.07262 -1.05982...	53.950935

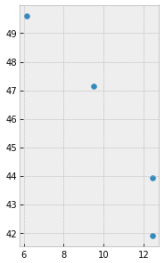
Indexing and Selecting Data

GeoPandas inherits the standard pandas methods for indexing/selecting data. This includes label based indexing with .loc and integer position based indexing with .iloc, which apply to both GeoSeries and GeoDataFrame objects. For more information on indexing/selecting, see the pandas documentation.

Selection by Index Position

Pandas provides a suite of methods in order to get purely integer based indexing. The semantics follow closely Python and NumPy slicing. These are 0-based indexing. When slicing, the start bound is included, while the upper bound is excluded. For instance `name = 'fudge'` with `name[0:3]` returns `'fud'`, where f is at 0 and g is at the 3 position with the upper bound excluded.

```
import matplotlib.pyplot as plt
plt.style.use('bmh') # better for plotting geometries vs general plots.

world = geopandas.read_file(geopandas.datasets.get_path('naturalearth_cities'))
northern_world = world.iloc[ 0:4 ]
northern_world.plot(figsize=(10,5))
plt.show()
```



Different choices for indexing

Object selection has had a number of user-requested additions in order to support more explicit location based indexing.

Getting values from an object with multi-axes selection uses the following notation (using `.loc` as an example, but the following applies to `.iloc` as well). Any of the axes accessors may be the null slice `:`. Axes left out of the specification are assumed to be `:`, e.g. `p.loc['a']` is equivalent to `p.loc['a', :, :]`.

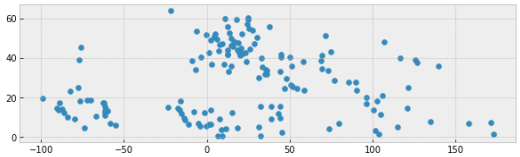
Object Type	Indexers
Series	`s.loc[indexer]`
DataFrame	`df.loc[row_indexer,column_indexer]`

Subset Points by Location

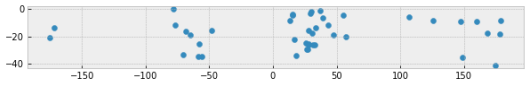
In addition to the standard pandas methods, GeoPandas also provides coordinate based indexing with the cx indexer, which slices using a bounding box. Geometries in the GeoSeries or GeoDataFrame that intersect the bounding box will be returned.

Using the world dataset, we can use this functionality to quickly select all cities in the northern and southern hemisphere using a `_CoordinateIndexer` using `.cx`. `.cx` allows you to quickly access the table's `geometry`, where indexing reflects `[x,y]` or `[lon,lat]`. Here we will query points above and below 0 degrees latitude:

```
world = geopandas.read_file(geopandas.datasets.get_path('naturalearth_cities'))
northern_world = world.cx[ : , 0: ]   # subsets all rows above 0 with a slice
northern_world.plot(figsize=(10, 5))
plt.show()
```




```
world = geopandas.read_file(geopandas.datasets.get_path('naturalearth_cities'))
southern_world = world.cx[ : , :0 ]   # subsets all rows below 0  with a slice
southern_world.plot(figsize=(10, 5))
plt.show()
```



Learning Objectives

- Do distance/proximity based analysis for points, lines, polygons
- Create buffers
- Get nearest neighbors

Review

- Spatial Vector Data
- Attributes & Indexing for Vector Data
- Creating Spatial Vector Data

Proximity Analysis - Buffers, Nearest Neighbor

In this chapter we are going to dig into some of the most common spatial operations. After this section you will be able to answer simple questions like "where is the nearest wendy's?", "Are there any homes within 50 yards of a highway?".

Buffer Analysis

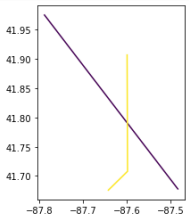
First, we will import the necessary modules and create two lines (click the + below to show code cell).

```
# Import modules
import pandas as pd
import geopandas as gpd
from shapely.geometry import Point, LineString, Polygon
from io import StringIO

data = """
ID,X,Y
1,  -87.789,  41.976
1,  -87.482,  41.677
2,  -87.599,  41.908
2,  -87.598,  41.708
2,  -87.643,  41.675
"""
# use StringIO to read in text chunk
df = pd.read_table(StringIO(data), sep=',')

#zip the coordinates into a point object and convert to a GeoData Frame
points = [Point(xy) for xy in zip(df.X, df.Y)]
points = gpd.GeoDataFrame(df, geometry=points, crs = 'EPSG:4326')
# create line for each ID
lines = points.groupby(['ID'])['geometry'].apply(lambda x:
LineString(x.tolist()))
lines = gpd.GeoDataFrame(lines, geometry='geometry', crs="EPSG:4326")
lines.reset_index(inplace=True)
```

Let's take a look at the data.

```
# plot county outline and add wells to axis (ax)
lines.plot(column='ID')
```

```
<AxesSubplot:>
```



Important

NEVER do distance analysis with unprojeted data (e.g. lat lon). Distances are best not measured in degrees! Instead use .to_crs() to convert it to a projected coordinate system with a linear unit in feet or meters etc.

Although it is not clearly stated the `distance` parameter is measured in the linear unit of the projection. So before we get started we need to make sure to use `to_crs()` to convert to a projected coordinate system.

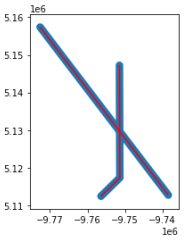
```
# plot county outline and add wells to axis (ax)
lines = lines.to_crs(3857)
# check the linear unit name in `unit_name`.
print(lines.crs.axis_info)
```

```
[Axis(name=Easting, abbrev=X, direction=east, unit_auth_code=EPSG, unit_code=9001,
unit_name=metre), Axis(name=Northing, abbrev=Y, direction=north,
unit_auth_code=EPSG, unit_code=9001, unit_name=metre)]
```

Starting from two lines we can start playing around with the buffer function. You can read the docs for the buffer function, unfortunately is split between two docs geopandas and shapely.

```
buf = lines.buffer(distance = 1000)
bp = buf.plot()
lines.plot(ax=bp, color='red')
print(buf)
```

```
0    POLYGON ((-9737657.488 5113325.388, -9737601.7...
1    POLYGON ((-9750364.761 5117342.850, -9750369.3...
dtype: geometry
```



Notice that we now have to polygon GEOMETRIES. This no longer has the line attributes associated with it. If we want to add back the attribute data we need to replace the original geometry column with new buffer geometry values.

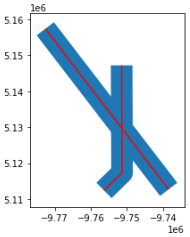
```
line_buffer = lines.copy()
line_buffer['geometry'] = buf
print(line_buffer)
```

```
   ID                                           geometry
0   1  POLYGON ((-9737657.488 5113325.388, -9737601.7...
1   2  POLYGON ((-9750364.761 5117342.850, -9750369.3...
```

There are a number of other parameters available to use, namely `cap_style`, and `single_sided`.

```
buf = lines.buffer(distance = 3000, cap_style = 2)
bp = buf.plot()
lines.plot(ax=bp, color='red')
```

```
<AxesSubplot:>
```



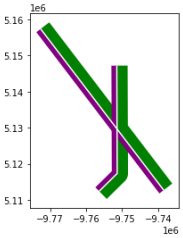
attribute	value
round	1
flat	2
square	3

Table 1 Buffer caps can be different shapes

We can also create left or right side buffers. Use negative distances for left, and positive values for right.

```
buf_right = lines.buffer(distance = 3000, single_sided = True)
bp = buf_right.plot(color='green')

buf_left = lines.buffer(distance = -1500, single_sided = True)
buf_left.plot(ax=bp, color='purple')
lines.plot(ax=bp, color='white')
```

```
<AxesSubplot:>
```



Nearest Neighbor Analysis

One commonly used GIS task is to be able to find the nearest neighbor. For instance, you might have a single Point object representing your home location, and then another set of locations representing e.g. public transport stops. Then, quite typical question is *"which of the stops is closest one to my home?"*

This is a typical nearest neighbor analysis, where the aim is to find the closest geometry to another geometry. [1]

In Python this kind of analysis can be done with shapely function called `nearest_points()` that returns a tuple of the nearest points in the input geometries.

Nearest point using Shapely

Let's start by testing how we can find the nearest Point using the `nearest_points` function of Shapely.

Let's create an origin Point and a few destination Points and find out the closest destination.

```
from shapely.geometry import Point, MultiPoint
from shapely.ops import nearest_points

orig = Point(1, 1.67)
dest1, dest2, dest3 = Point(0, 1.45), Point(2, 2), Point(0, 2.5)
```

To be able to find out the closest destination point from the origin, we need to create a MultiPoint object from the destination points.

```
destinations = MultiPoint([dest1, dest2, dest3])
print(destinations)
```

```
MULTIPOINT (0 1.45, 2 2, 0 2.5)
```

Okey, now we can see that all the destination points are represented as a single MultiPoint object. Now we can find out the nearest destination point by using `nearest_points()` function.

```
nearest_geoms = nearest_points(orig, destinations)
original_point, nearest_destination = nearest_geoms
print(nearest_geoms)
print('Coordinates of original point:',original_point)
print('Coordinates of closest destination point:',nearest_destination)
```

```
(<shapely.geometry.point.Point object at 0x7ff96043f750>,
<shapely.geometry.point.Point object at 0x7ff96043fe90>)
Coordinates of original point: POINT (1 1.67)
Coordinates of closest destination point: POINT (0 1.45)
```

As you can see the `nearest_points` function returns a tuple of geometries where the first item is the geometry of our origin point and the second item (at index 1) is the actual nearest geometry from the destination points. Hence, the closest destination point seems to be the one located at coordinates (0, 1.45).

This is the basic logic how we can find the nearest point from a set of points.

Nearest points using Geopandas

Of course, the previous example is not really useful yet. Hence, next I show, how it is possible to find nearest points from a set of origin points to a set of destination points using GeoDataFrames. In this example we will recreate the previous example but use geopandas, however this data could come from any shapefile.

- First we need to create a function that takes advantage of the previous function but is tailored to work with two GeoDataFrames.

```
from shapely.ops import nearest_points

def _nearest(row, df1, df2, geom1='geometry', geom2='geometry', df2_column=None):
    """Find the nearest point and return the corresponding value from specified
column."""

    # create object usable by Shapely
    geom_union = df2.unary_union

    # Find the geometry that is closest
    nearest = df2[geom2] == nearest_points(row[geom1], geom_union)[1]
    # Get the corresponding value from df2 (matching is based on the geometry)
    if df2_column is None:
        value = df2[nearest].index[0]
    else:
        value = df2[nearest][df2_column].values[0]
    return value

def nearest(df1, df2, geom1_col='geometry', geom2_col='geometry',
df2_column=None):
    """Find the nearest point and return the corresponding value from specified
column.
    :param df1: Origin points
    :type df1: geopandas.GeoDataFrame
    :param df2: Destination points
    :type df2: geopandas.GeoDataFrame
    :param geom1_col: name of column holding coordinate geometry, defaults to
'geometry'
    :type geom1_col: str, optional
    :param geom2_col: name of column holding coordinate geometry, defaults to
'geometry'
    :type geom2_col: str, optional
    :param df2_column: column name to return from df2, defaults to None
    :type df2_column: str, optional
    :return: df1 with nearest neighbor index or df2_column appended
    :rtype: geopandas.GeoDataFrame
    """
    df1['nearest_id'] = df1.apply(_nearest,  df1=df1, df2=df2,
                                  geom1=geom1_col, geom2=geom2_col,
                                  df2_column=df2_column, axis=1)
    return df1
```

```
# generate origin and destination points as geodataframe
orig = {'name': ['Origin_1','Origin_2'],
     'geometry': [Point(-77.3,38.94),Point(-77.41,39.93)]}
orig = gpd.GeoDataFrame(orig, crs="EPSG:4326")
print(orig)

dest = {'name': ['Baltimore','Washington', 'Fredrick'],
     'geometry': [ Point(-76.61,39.29,), Point(-77.04,38.91),
Point(-77.40,39.41)]}
dest = gpd.GeoDataFrame(dest, crs="EPSG:4326")
print(dest)
```

```
      name                   geometry
0  Origin_1  POINT (-77.30000 38.94000)
1  Origin_2  POINT (-77.41000 39.93000)
         name                   geometry
0   Baltimore  POINT (-76.61000 39.29000)
1  Washington  POINT (-77.04000 38.91000)
2    Fredrick  POINT (-77.40000 39.41000)
```

Okay now we are ready to use our function and find closest Points (taking the value from id column) from df2 to df1 centroids

```
nearest = nearest(df1=orig, df2=dest, df2_column='name')
nearest.head()
```

	name	geometry	nearest_id
0	Origin_1	POINT (-77.30000 38.94000)	Washington
1	Origin_2	POINT (-77.41000 39.93000)	Fredrick

That's it! Now we found the closest point for each centroid and got the `index` value or column name from our addresses into the `df1` GeoDataFrame.

> **Note**
>
> If you want to do nearest neighbor analysis with polygons, you can simply use the centroid. If you have a geopandas polygon called poly, run poly['centroid'] = poly.centroid to store the centroid values in the attribute table.

Sources

[1] automating-gis-processes

Merge Data & Dissolve Polygons

By: Steven Chao

> **Learning Objectives**
>
> - Import dataframes into Python for analysis
> - Perform basic dataframe column operations
> - Merge dataframes using a unique key
> - Group attributes based on a similar attribute
> - Dissolve vector geometries based on attribute values

> **Review**
>
> - Data Structures
> - Vector Data

Introduction

Dataframes are widely used in Python for manipulating data. Recall that a dataframe is essentially an Excel spreadsheet (a 2-D table of rows and columns); in fact, many of the functions that you might use in Excel can often be replicated when working with dataframes in Python!

This chapter will introduce you to some of the basic operations that you can perform on dataframes. We will use these basic operations in order to calculate and map poverty rates in the Commonwealth of Virginia. We will pull data from the US Census Bureau's American Community Survey (ACS) 2019 (see this page for the data).

```
# Import modules
import matplotlib.pyplot as plt
import pandas as pd
import geopandas as gpd
from census import Census
from us import states
```

Accessing Data

Import census data

Let's begin by accessing and importing census data. Importing census data into Python requires a Census API key. A key can be obtained from Census API Key. **It will provide you with a unique 40 digit text string. Please keep track of this number. Store it in a safe place.**

```
# Set API key
c = Census("CENSUS API KEY HERE")
```

```
#ignore this, I am just reading in my api key privately
with open("../../../census_api.txt", "r") as f:
    c = Census(f.read().replace('\n', ''))
```

With the Census API key set, we will access the census data at the tract level for the Commonwealth of Virginia from the 2019 ACS, specifically the `ratio of income to poverty in the past 12 months` (`C17002_001E`, total; `C17002_002E`, < 0.50; and `C17002_003E`, 0.50 - 0.99) variables and the `total population` (`B01003_001E`) variable. For more information on why these variables are used, refer to the US Census Bureau's article on how the Census Bureau measures poverty and the list of variables found in ACS.

The `census` package provides us with some easy convenience methods that allow us to obtain data for a wide variety of geographies. The FIPS code for Virginia is 51, but if needed, we can also use the `us` library to help us figure out the relevant FIPS code.

```
# Obtain Census variables from the 2019 ACS at the tract level for the
Commonwealth of Virginia (FIPS code: 51)
# C17002_001E: count of ratio of income to poverty in the past 12 months (total)
# C17002_002E: count of ratio of income to poverty in the past 12 months (< 0.50)
# C17002_003E: count of ratio of income to poverty in the past 12 months (0.50 -
0.99)
# B01003_001E: total population
# Sources: https://api.census.gov/data/2019/acs/acs5/variables.html;
https://pypi.org/project/census/
va_census = c.acs5.state_county_tract(fields = ('NAME', 'C17002_001E',
'C17002_002E', 'C17002_003E', 'B01003_001E'),
                                      state_fips = states.VA.fips,
                                      county_fips = "*",
                                      tract = "*",
                                      year = 2017)
```

Now that we have accessed the data and assigned it to a variable, we can read the data into a dataframe using the `pandas` library.

```
# Create a dataframe from the census data
va_df = pd.DataFrame(va_census)

# Show the dataframe
print(va_df.head(2))
print('Shape: ', va_df.shape)
```

```
                                      NAME  C17002_001E  C17002_002E  \
0     Census Tract 60, Norfolk city, Virginia       3947.0        284.0
1  Census Tract 65.02, Norfolk city, Virginia       3287.0        383.0

   C17002_003E  B01003_001E state county   tract
0        507.0       3947.0    51    710  006000
1        480.0       3302.0    51    710  006502
Shape:  (1907, 8)
```

By showing the dataframe, we can see that there are 1907 rows (therefore 1907 census tracts) and 8 columns.

Import Shapefile

Let's also read into Python a shapefile of the Virginia census tracts and reproject it to the UTM Zone 17N projection. (This shapefile can be downloaded on the Census Bureau's website on the Cartographic Boundary Files page or the TIGER/Line Shapefiles page.)

```
# Access shapefile of Virginia census tracts
va_tract =
gpd.read_file("https://www2.census.gov/geo/tiger/TIGER2019/TRACT/tl_2019_51_tract.
zip")

# Reproject shapefile to UTM Zone 17N
# https://spatialreference.org/ref/epsg/wgs-84-utm-zone-17n/
va_tract = va_tract.to_crs(epsg = 32617)

# Print GeoDataFrame of shapefile
print(va_tract.head(2))
print('Shape: ', va_tract.shape)

# Check shapefile projection
print("\nThe shapefile projection is: {}".format(va_tract.crs))
```

```
  STATEFP COUNTYFP TRACTCE        GEOID    NAME             NAMELSAD  MTFCC  \
0      51      700  032132  51700032132  321.32  Census Tract 321.32  G5020
1      51      700  032226  51700032226  322.26  Census Tract 322.26  G5020

  FUNCSTAT    ALAND  AWATER     INTPTLAT      INTPTLON  \
0        S  2552457       0  +37.1475176  -076.5212499
1        S  3478916  165945  +37.1625163  -076.5527816

                                      geometry
0  POLYGON ((897174.191 4119897.084, 897174.811 4...
1  POLYGON ((893470.562 4123469.385, 893542.722 4...
Shape:  (1907, 13)

The shapefile projection is: epsg:32617
```

By printing the shapefile, we can see that the shapefile also has 1907 rows (1907 tracts). This number matches with the number of census records that we have on file. Perfect!

Not so fast, though. We have a potential problem: We have the census data, and we have the shapefile of census tracts that correspond with that data, but they are stored in two separate variables (`va_df` and `va_tract` respectively)! That makes it a bit difficult to map since these two separate datasets aren't connected to each other.

Performing Dataframe Operations

Create new column from old columns to get combined FIPS code

To solve this problem, we can join the two dataframes together via a field or column that is common to both dataframes, which is referred to as a key.

Looking at the two datasets above, it appears that the `GEOID` column from `va_tract` and the `state`, `county`, and `tract` columns combined from `va_df` could serve as the unique key for joining these two dataframes together. In their current forms, this join will not be successful, as we'll need to merge the `state`, `county`, and `tract` columns from `va_df` together to make it parallel to the `GEOID` column from `va_tract`. We can simply add the columns together, much like math or the basic operators in Python, and assign the "sum" to a new column.

To create a new column–or call an existing column in a dataframe–we can use indexing with `[]` and the column name (string). (There is a different way if you want to access columns using the index number; read more about indexing and selecting data in the pandas documentation.)

```
# Combine state, county, and tract columns together to create a new string and
assign to new column
va_df["GEOID"] = va_df["state"] + va_df["county"] + va_df["tract"]
```

Printing out the first rew rows of the dataframe, we can see that the new column `GEOID` has been created with the values from the three columns combined.

```
# Print head of dataframe
va_df.head(2)
```

	NAME	C17002_001E	C17002_002E	C17002_003E	B01003_001E	state	county	tract	GEOID
0	Census Tract 60, Norfolk city, Virginia	3947.0	284.0	507.0	3947.0	51	710	006000	51710006000
1	Census Tract 65.02, Norfolk city, Virginia	3287.0	383.0	480.0	3302.0	51	710	006502	51710006502

Remove dataframe columns that are no longer needed

To reduce clutter, we can delete the `state`, `county`, and `tract` columns from `va_df` since we don't need them anymore. Remember that when we want to modify a dataframe, we must assign the modified dataframe back to the original variable (or a new one, if preferred). Otherwise, any modifications won't be saved. An alternative to assigning the dataframe back to the variable is to simply pass `inplace = True`. For more information, see the pandas help documentation on **`drop`**.

```
# Remove columns
va_df = va_df.drop(columns = ["state", "county", "tract"])

# Show updated dataframe
va_df.head(2)
```

	NAME	C17002_001E	C17002_002E	C17002_003E	B01003_001E	GEOID
0	Census Tract 60, Norfolk city, Virginia	3947.0	284.0	507.0	3947.0	51710006000
1	Census Tract 65.02, Norfolk city, Virginia	3287.0	383.0	480.0	3302.0	51710006502

Check column data types

The key in both dataframe must be of the same data type. Let's check the data type of the `GEOID` columns in both dataframes. If they aren't the same, we will have to change the data type of columns to make them the same.

```
# Check column data types for census data
print("Column data types for census data:\n{}".format(va_df.dtypes))

# Check column data types for census shapefile
print("\nColumn data types for census shapefile:\n{}".format(va_tract.dtypes))

# Source: https://pandas.pydata.org/pandas-
docs/stable/reference/api/pandas.DataFrame.dtypes.html
```

```
Column data types for census data:
NAME           object
C17002_001E    float64
C17002_002E    float64
C17002_003E    float64
B01003_001E    float64
GEOID           object
dtype: object

Column data types for census shapefile:
STATEFP       object
COUNTYFP      object
TRACTCE       object
GEOID         object
NAME          object
NAMELSAD      object
MTFCC         object
FUNCSTAT      object
ALAND          int64
AWATER         int64
INTPTLAT      object
INTPTLON      object
geometry    geometry
dtype: object
```

Looks like the `GEOID` columns are the same!

Merge dataframes

Now, we are ready to merge the two dataframes together, using the `GEOID` columns as the primary key. We can use the `merge` method in `GeoPandas` called on the `va_tract` shapefile dataset.

```
# Join the attributes of the dataframes together
# Source: https://geopandas.org/docs/user_guide/mergingdata.html
va_merge = va_tract.merge(va_df, on = "GEOID")

# Show result
print(va_merge.head(2))
print('Shape: ', va_merge.shape)
```

```
  STATEFP COUNTYFP TRACTCE        GEOID  NAME_x            NAMELSAD  MTFCC  \
0      51      700  032132  51700032132  321.32  Census Tract 321.32  G5020
1      51      700  032226  51700032226  322.26  Census Tract 322.26  G5020

  FUNCSTAT    ALAND  AWATER     INTPTLAT      INTPTLON  \
0        S  2552457       0  +37.1475176  -076.5212499
1        S  3478916  165945  +37.1625163  -076.5527816

                                      geometry  \
0  POLYGON ((897174.191 4119897.084, 897174.811 4...
1  POLYGON ((893470.562 4123469.385, 893542.722 4...

                                        NAME_y  C17002_001E  C17002_002E  \
0  Census Tract 321.32, Newport News city, Virginia       5025.0        161.0
1  Census Tract 322.26, Newport News city, Virginia       4167.0        736.0

   C17002_003E  B01003_001E
0        342.0       5079.0
1        559.0       4167.0
Shape:  (1907, 18)
```

Success! We still have 1907 rows, which means that all rows (or most of them) were successfully matched! Notice how the census data has been added on after the shapefile data in the dataframe.

Some additional notes about joining dataframes:

- the columns for the key do not need to have the same name.
- for this join, we had a one-to-one relationship, meaning one attribute in one dataframe matched to one (and only one) attribute in the other dataframe. Joins with a many-to-one, one-to-many, or many-to-many relationship are also possible, but in some cases, they require some special considerations. See this Esri ArcGIS help documentation on joins and relates for more information.

Subset dataframe

Now that we merged the dataframes together, we can further clean up the dataframe and remove columns that are not needed. Instead of using the `drop` method, we can simply select the columns we want to keep and create a new dataframe with those selected columns.

```
# Create new dataframe from select columns
va_poverty_tract = va_merge[["STATEFP", "COUNTYFP", "TRACTCE", "GEOID",
"geometry", "C17002_001E", "C17002_002E", "C17002_003E", "B01003_001E"]]

# Show dataframe
print(va_poverty_tract.head(2))
print('Shape: ', va_poverty_tract.shape)
```

```
  STATEFP COUNTYFP TRACTCE        GEOID  \
0      51      700  032132  51700032132
1      51      700  032226  51700032226

                                            geometry  C17002_001E  \
0  POLYGON ((897174.191 4119897.084, 897174.811 4...       5025.0
1  POLYGON ((893470.562 4123469.385, 893542.722 4...       4167.0

   C17002_002E  C17002_003E  B01003_001E
0        161.0        342.0       5079.0
1        736.0        559.0       4167.0
Shape:  (1907, 9)
```

Notice how the number of columns dropped from 13 to 9.

Dissolve geometries and get summarized statistics to get poverty statistics at the county level

Next, we will group all the census tracts within the same county (`COUNTYFP`) and aggregate the poverty and population values for those tracts within the same county. We can use the `dissolve` function in `GeoPandas`, which is the spatial version of `groupby` in `pandas`. We use `dissolve` instead of `groupby` because the former also groups and merges all the geometries (in this case, census tracts) within a given group (in this case, counties).

```
# Dissolve and group the census tracts within each county and aggregate all the
values together
# Source: https://geopandas.org/docs/user_guide/aggregation_with_dissolve.html
va_poverty_county = va_poverty_tract.dissolve(by = 'COUNTYFP', aggfunc = 'sum')

# Show dataframe
print(va_poverty_county.head(2))
print('Shape: ', va_poverty_county.shape)
```

```
                                                geometry  C17002_001E  \
COUNTYFP
001       POLYGON ((971901.668 4160101.088, 971814.409 4...      32345.0
003       POLYGON ((734957.267 4207640.156, 734931.249 4...      97587.0

          C17002_002E  C17002_003E  B01003_001E
COUNTYFP
001            2423.0       3993.0      32840.0
003            5276.0       4305.0     105105.0
Shape:  (133, 5)
```

Notice that we got the number of rows down from 1907 to 133.

Perform column math to get poverty rates

We can estimate the poverty rate by dividing the sum of `C17002_002E` (ratio of income to poverty in the past 12 months, < 0.50) and `C17002_003E` (ratio of income to poverty in the past 12 months, 0.50 - 0.99) by `B01003_001E` (total population).

Side note: Notice that `C17002_001E` (ratio of income to poverty in the past 12 months, total), which theoretically should count everyone, does not exactly match up with `B01003_001E` (total population). We'll disregard this for now since the difference is not too significant.

```
# Get poverty rate and store values in new column
va_poverty_county["Poverty_Rate"] = (va_poverty_county["C17002_002E"] +
va_poverty_county["C17002_003E"]) / va_poverty_county["B01003_001E"] * 100

# Show dataframe
va_poverty_county.head(2)
```

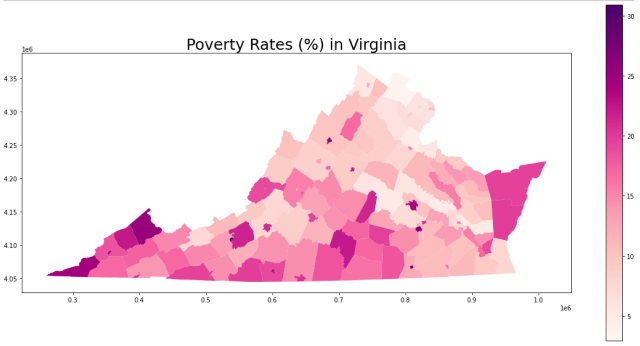
COUNTYFP	geometry	C17002_001E	C17002_002E	C17002_003E	B01003_001E	Poverty_Rate
001	POLYGON ((971901.668 4160101.088, 971814.409 4...	32345.0	2423.0	3993.0	32840.0	19.537150
003	POLYGON ((734957.267 4207640.156, 734931.249 4...	97587.0	5276.0	4305.0	105105.0	9.115646

Plotting Results

Finally, since we have the spatial component connected to our census data, we can plot the results!

```
# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (20, 10))

# Plot data
# Source: https://geopandas.readthedocs.io/en/latest/docs/user_guide/mapping.html
va_poverty_county.plot(column = "Poverty_Rate",
                       ax = ax,
                       cmap = "RdPu",
                       legend = True)

# Stylize plots
plt.style.use('bmh')

# Set title
ax.set_title('Poverty Rates (%) in Virginia', fontdict = {'fontsize': '25',
'fontweight' : '3'})
```

```
Text(0.5, 1.0, 'Poverty Rates (%) in Virginia')
```



Learning Objectives

- Clip a vector feature using another vector feature
- Select features by their attributes
- Select features by their locations

Review

- Spatial Vector Data
- Attributes & Indexing for Vector Data
- Creating Spatial Vector Data

Extracting Spatial Data

Subsetting and extracting data is useful when we want to select or analyze a portion of the dataset based on a feature's location, attribute, or its spatial relationship to another dataset.

In this chapter, we will explore three ways that data from a GeoDataFrame can be subsetted and extracted: clip, select location by attribute, and select by location.

Setup

First, let's import the necessary modules (click the + below to show code cell).

```
# Import modules
import geopandas as gpd
import matplotlib.pyplot as plt
import pandas as pd
from shapely.geometry import Polygon
```

We will utilize shapefiles of San Francisco Bay Area county boundaries and wells within the Bay Area and the surrounding 50 km. We will first load in the data and reproject the data (click the + below to show code cell).

Important

All the data must have the same coordinate system in order for extraction to work correctly.

```
# Load data

# County boundaries
# Source: https://opendata.mtc.ca.gov/datasets/san-francisco-bay-region-counties-clipped?geometry=-125.590%2C37.123%2C-119.152%2C38.640
counties =
gpd.read_file("../_static/e_vector_shapefiles/sf_bay_counties/sf_bay_counties.shp"
)

# Well locations
# Source: https://gis.data.ca.gov/datasets/3a3e681b894644a9a95f9815aeeeb57f_0?geometry=-123.143%2C36.405%2C-119.230%2C37.175
# Modified by author so that only the well locations within the counties and the surrounding 50 km were kept
wells =
gpd.read_file("../_static/e_vector_shapefiles/sf_bay_wells_50km/sf_bay_wells_50km.shp")

# Reproject data to NAD83(HARN) / California Zone 3
# https://spatialreference.org/ref/epsg/2768/
proj = 2768
counties = counties.to_crs(proj)
wells = wells.to_crs(proj)
```

We will also create a rectangle over a part of the Bay Area. We have identified coordinates to use for this rectangle, but you can also use bbox finder to generate custom bounding boxes and obtain their coordinates (click the + below to show code cell).

```
# Create list of coordinate pairs
coordinates = [[1790787, 736108], [1929652, 736108], [1929652, 598414], [1790787, 598414]]

# Create a Shapely polygon from the coordinate-tuple list
poly_shapely = Polygon(coordinates)

# Create a dictionary with needed attributes and required geometry column
attributes_df = {'Attribute': ['name1'], 'geometry': poly_shapely}

# Convert shapely object to a GeoDataFrame
poly = gpd.GeoDataFrame(attributes_df, geometry = 'geometry', crs = "EPSG:2768")
```

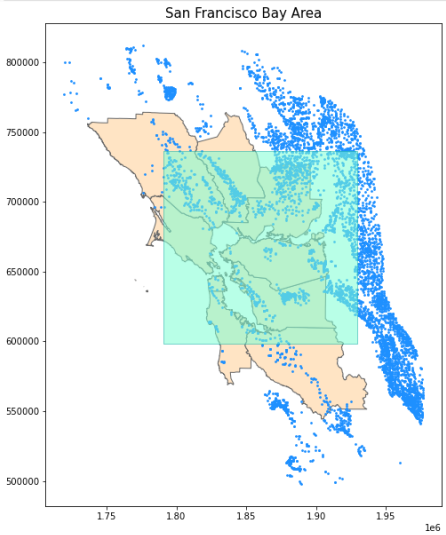
We'll define some functions to make displaying and mapping our results a bit easier (click the + below to show code cell).

```
def display_table(table_name, attribute_table):
    '''Display the first and last five rows of attribute table.'''

    # Print title
    print("Attribute Table: {}".format(table_name))

    # Print number of rows and columns
    print("\nTable shape (rows, columns): {}".format(attribute_table.shape))

    # Display first two rows of attribute table
    print("\nFirst two rows:")
    display(attribute_table.head(2))

    # Display last two rows of attribute table
    print("\nLast two rows:")
    display(attribute_table.tail(2))


def plot_df(result_name, result_df, result_geom_type, area = None):
    '''Plot the result on a map and add the outlines of the original
shapefiles.'''

    # Create subplots
    fig, ax = plt.subplots(1, 1, figsize = (10, 10))

    # Plot data depending on vector type
    # For points
    if result_geom_type == "point":

        # Plot data
        counties.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
        wells.plot(ax = ax, marker = 'o', color = 'dimgray', markersize = 3)
        result_df.plot(ax = ax, marker = 'o', color = 'dodgerblue', markersize =
3)

    # For polygons
    else:

        # Plot overlay data
        result_df.plot(ax = ax, cmap = 'Set2', edgecolor = 'black')

        # Plot outlines of original shapefiles
        counties.plot(ax = ax, color = 'none', edgecolor = 'dimgray')

    # Add additional outlined boundary if specified
    if area is not None:

        # Plot data
        area.plot(ax = ax, color = 'none', edgecolor = 'lightseagreen', linewidth
= 3)

    # Else, pass
    else:
        pass

    # Stylize plots
    plt.style.use('bmh')

    # Set title
    ax.set_title(result_name, fontdict = {'fontsize': '15', 'fontweight' : '3'})
```

Let's take a look at what our input data looks like.

```
# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (10, 10))

# Plot data
counties.plot(ax = ax, color = 'bisque', edgecolor = 'dimgray')
wells.plot(ax = ax, marker = 'o', color = 'dodgerblue', markersize = 3)
poly.plot(ax = ax, color = 'aquamarine', edgecolor = 'lightseagreen', alpha =
0.55)

# Stylize plots
plt.style.use('bmh')

# Set title
ax.set_title('San Francisco Bay Area', fontdict = {'fontsize': '15', 'fontweight'
: '3'})
```

```
Text(0.5, 1.0, 'San Francisco Bay Area')
```



Clip Spatial Polygons

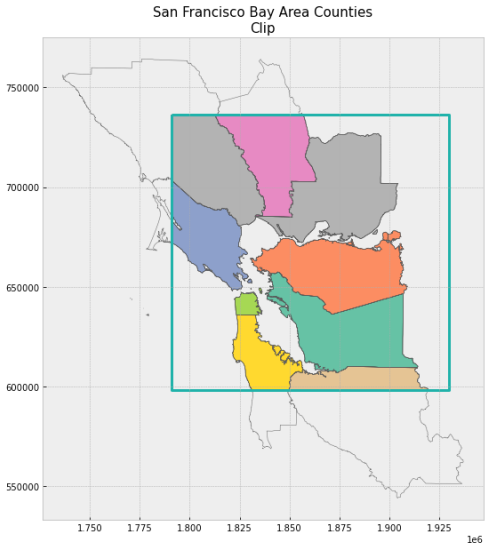
Clip extracts and keeps only the geometries of a vector feature that are within extent of another vector feature (think of it like a cookie-cutter or mask). We can use `clip()` in `geopandas`, with the first parameter being the vector that will be clipped and the second parameter being the vector that will define the extent of the clip. *All attributes for the resulting clipped vector will be kept.* [1]

> **Note**
>
> This function is only available in the more recent versions of `geopandas`.

We will first clip the Bay Area counties polygon to our created rectangle polygon.

```
# Clip data
clip_counties = gpd.clip(counties, poly)

# Display attribute table
display(clip_counties)

# Plot clip
plot_df(result_name = "San Francisco Bay Area Counties\nClip", result_df =
clip_counties, result_geom_type = "polygon", area = poly)
```

	coname	geometry
0	Alameda County	MULTIPOLYGON (((1860234.837 612219.122, 186007...
1	Contra Costa County	MULTIPOLYGON (((1836501.066 656512.837, 183649...
2	Marin County	MULTIPOLYGON (((1822277.787 655539.623, 182231...
3	Napa County	POLYGON ((1860171.549 724651.231, 1860170.736 ...
4	San Francisco County	MULTIPOLYGON (((1834364.295 641524.540, 183435...
5	San Mateo County	MULTIPOLYGON (((1850042.437 615031.357, 185003...
6	Santa Clara County	MULTIPOLYGON (((1858733.082 607586.450, 185874...
7	Solano County	MULTIPOLYGON (((1875491.908 673077.900, 187549...
8	Sonoma County	POLYGON ((1813621.641 736108.000, 1813605.098 ...

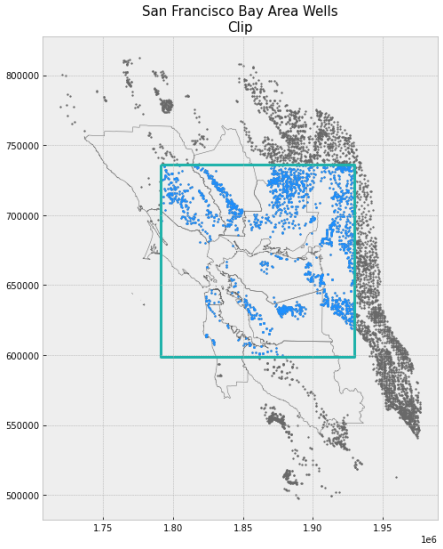


We can clip any vector type. Next, we will clip the wells point data to our created rectangle polygon.

```
# Clip data
clip_wells = gpd.clip(wells, poly)

# Display attribute table
display(clip_wells)

# Plot clip
plot_df(result_name = "San Francisco Bay Area Wells\nClip", result_df =
clip_wells, result_geom_type = "point", area = poly)
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
2	Flag City Well 3	Unknown	Single Well	0	POINT (1922067.711 679380.918)
14	Flag City Well 1	Unknown	Single Well	170	POINT (1922680.537 679641.516)
15	Flag City Well 2	Unknown	Single Well	180	POINT (1921777.555 679393.668)
25	W-041	Other	Single Well	256	POINT (1919777.393 713390.175)
26	W-042	Other	Single Well	245	POINT (1917586.004 713468.595)
...	...	...	...	...	...
6029	None	Unknown	Unknown	465	POINT (1848120.290 703766.303)
6030	None	Irrigation	Unknown	51	POINT (1842158.280 695286.242)
6031	None	Unknown	Unknown	315	POINT (1848812.488 704308.579)
6032	GV FARM	Residential	Single Well	67	POINT (1855297.377 693189.634)
6033	None	Residential	Unknown	76	POINT (1841755.238 707939.225)

2013 rows × 5 columns



Select Location by Attributes

Selecting by attribute selects only the features in a dataset whose attribute values match the specified criteria. `geopandas` uses the indexing and selection methods in `pandas`, so data in a GeoDataFrame can be selected and queried in the same way a `pandas` dataframe can. [2], [3], [4]

We will use use different criteria to subset the wells dataset.

```
# Display attribute table
display(wells.head(2))
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
0	2400064-001	Public Supply	Single Well	0	POINT (1968290.923 592019.918)
1	2400099-001	Public Supply	Single Well	0	POINT (1969113.543 595876.691)

The criteria can use a variety of operators, including comparison and logical operators.

```
# Select wells that are public supply
wells_public = wells[(wells["WELL_USE"] == "Public Supply")]

# Display first two and last two rows of attribute table
display_table(table_name = "San Francisco Bay Area Wells - Public Supply",
attribute_table = wells_public)
```

```
Attribute Table: San Francisco Bay Area Wells - Public Supply

Table shape (rows, columns): (33, 5)

First two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
0	2400064-001	Public Supply	Single Well	0	POINT (1968290.923 592019.918)
1	2400099-001	Public Supply	Single Well	0	POINT (1969113.543 595876.691)

```
Last two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
3368	Aptos Creek PW	Public Supply	Single Well	713	POINT (1875127.938 553764.966)
3369	Country Club PW	Public Supply	Single Well	495	POINT (1877219.818 552137.819)

```
# Select wells that are public supply and have a depth greater than 50 ft
wells_public_deep = wells[(wells["WELL_USE"] == "Public Supply") &
(wells["WELL_DEPTH"] > 50)]

# Display first two and last two rows of attribute table
display_table(table_name = "San Francisco Bay Area Wells - Public Supply with
Depth Greater than 50 ft", attribute_table = wells_public_deep)
```

```
Attribute Table: San Francisco Bay Area Wells - Public Supply with Depth Greater
than 50 ft

Table shape (rows, columns): (24, 5)

First two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
919	Granite Way PW	Public Supply	Single Well	670	POINT (1875403.340 554078.279)
1082	Aptos Jr. High 2 PW	Public Supply	Single Well	590	POINT (1876872.389 553821.138)

```
Last two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
3368	Aptos Creek PW	Public Supply	Single Well	713	POINT (1875127.938 553764.966)
3369	Country Club PW	Public Supply	Single Well	495	POINT (1877219.818 552137.819)

```
# Select wells that are public supply and have a depth greater than 50 ft OR are
residential
wells_public_deep_residential = wells[((wells["WELL_USE"] == "Public Supply") &
(wells["WELL_DEPTH"] > 50)) | (wells["WELL_USE"] == "Residential")]

# Display first two and last two rows of attribute table
display_table(table_name = "San Francisco Bay Area Wells - Public Supply with
Depth Greater than 50 ft or Residential", attribute_table =
wells_public_deep_residential)
```

```
Attribute Table: San Francisco Bay Area Wells - Public Supply with Depth Greater
than 50 ft or Residential

Table shape (rows, columns): (725, 5)

First two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
137	None	Residential	Unknown	78	POINT (1886296.744 730378.969)
138	None	Residential	Unknown	80	POINT (1877320.295 730464.652)

```
Last two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
6032	GV FARM	Residential	Single Well	67	POINT (1855297.377 693189.634)
6033	None	Residential	Unknown	76	POINT (1841755.238 707939.225)

Select by Location

Selecting by location selects features based on its relative spatial relationship with another dataset. In other words, features are selected based on their location relative to the location of another dataset.

For example:

- to know how many wells there are in Santa Clara County, we could select all the wells that fall within Santa Clara County boundaries (which we do in one of the examples below)
- to know what rivers flow through Santa Clara County, we could select all the river polylines that intersect with Santa Clara County boundaries

For more information on selecting by location and spatial relationships, check out this ArcGIS documentation.

There are multiple spatial relationships available in `geopandas`: [5]

Spatial Relationship	Function(s)	Description
contains	`contains()`	geometry encompasses the other geometry's boundary and interior without any boundaries touching
covers	`covers()`	all of the geometry's points are to the exterior of the other geometry's points
covered by	`covered_by()`	all of the geometry's points are to the interior of the other geometry's points
crosses	`crosses()`	geometry's interior intersects that of the other geometry, provided that the geometry does not contain the other and the dimension of the intersection is less than the dimension of either geometry
disjoint	`disjoint()`	geometry's boundary and interior do not intersect the boundary and interior of the other geometry
equal geometry	`geom_equals()`, `geom_almost_equals()`, `geom_equals_exact()`	geometry's boundary, interior, and exterior matches (within a range) those of the other
intersects	`intersects()`	geometry's boundary or interior touches or crosses any part of the other geometry
overlaps	`overlaps()`	geometry shares at least one point, but not all points, with the other geometry, provided that the geometries and the intersection of their interiors all have the same dimensions
touches	`touches()`	geometry shares at least one point with the other geometry, provided that no part of the geometry's interior intersects with the other geometry
within	`within()`	geometry is enclosed in the other geometry (geometry's boundary and interior intersects with the other geometry's interior only)

Note

The functions for these spatial relationships will generally output a `pandas` series with Boolean values (`True` or `False`) whose indexing corresponds with the input dataset (from where we want to subset the data). We can use these Boolean values to subset the dataset (where only the rows that have a `True` output will be retained).

Method 1 - Shapely vector

These functions can be used to select features that have the specified spatial relationship with a single Shapely vector.

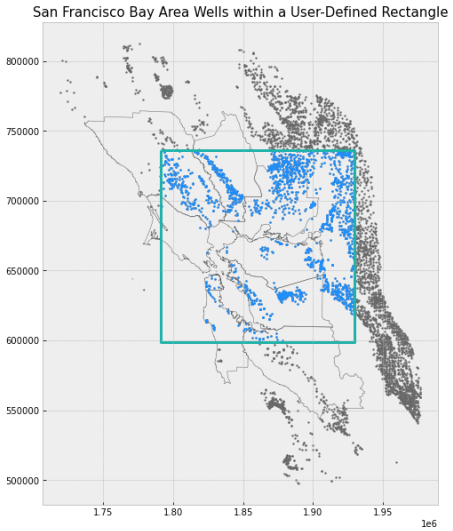
```
# Select wells that fall within Shapely rectangle
wells_within_rect_shapely = wells[wells.within(poly_shapely)]

# Display first two and last two rows of attribute table
display_table(table_name = "San Francisco Bay Area Wells within a User-Defined
Rectangle", attribute_table = wells_within_rect_shapely)

# Plot selection
plot_df(result_name = "San Francisco Bay Area Wells within a User-Defined
Rectangle", result_df = wells_within_rect_shapely, result_geom_type = "point",
area = poly)
```

```
Attribute Table: San Francisco Bay Area Wells within a User-Defined Rectangle

Table shape (rows, columns): (2013, 5)

First two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
2	Flag City Well 3	Unknown	Single Well	0	POINT (1922067.711 679380.918)
14	Flag City Well 1	Unknown	Single Well	170	POINT (1922680.537 679641.516)

```
Last two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
6032	GV FARM	Residential	Single Well	67	POINT (1855297.377 693189.634)
6033	None	Residential	Unknown	76	POINT (1841755.238 707939.225)



Method 2 - GeoDataFrame

If we're trying to select features that have a specified spatial relationship with another `geopandas` object, it gets a little tricky. This is because the `geopandas` spatial relationship functions verify the spatial relationship either row by row or index by index. In other words, the first row in the first dataset will be compared with the corresponding row or index in the second dataset, the second row in the first dataset will be compared with the corresponding row or index in the second dataset, and so on. [6], [7]

As a result, the number of rows need to correspond or the indices numbers need to match between the two datasets–or else we'll get a warning and the output will be empty.

Because each record in a GeoDataFrame has a geometry column that stores that record's geometry as a `shapely` object, we can call this object if we want to check a bunch of features against one extent (with one geometry). [6], [7]

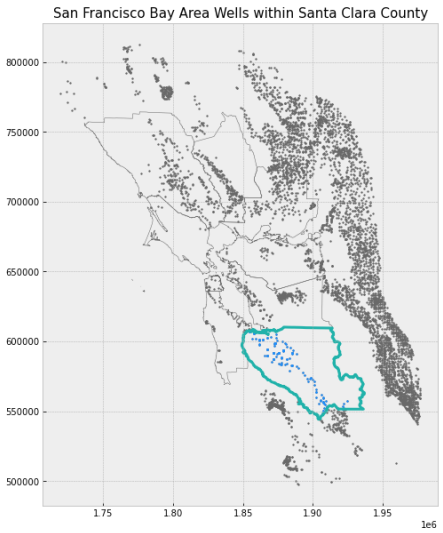
```
# Select the Santa Clara County boundary
sc_county = counties[counties["coname"] == "Santa Clara County"]

# Subset the GeoDataFrame by checking which wells are within Santa Clara County's
Shapely object
wells_within_sc_shapely = wells[wells.within(sc_county.geometry.values[0])]

# Display first two and last two rows of attribute table
display_table(table_name = "San Francisco Bay Area Wells within Santa Clara
County", attribute_table = wells_within_sc_shapely)

# Plot selection
plot_df(result_name = "San Francisco Bay Area Wells within Santa Clara County",
result_df = wells_within_sc_shapely, result_geom_type = "point", area = sc_county)
```

```
Attribute Table: San Francisco Bay Area Wells within Santa Clara County

Table shape (rows, columns): (136, 5)

First two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
980	CalFire Pacheco No 2	Residential	Single Well	0	POINT (1924674.853 557296.117)
1020	05S02W35R002	Observation	Part of a nested/multi-completion well	80	POINT (1863211.218 606236.916)

```
Last two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
5962	07S01W25L001	Observation	Single Well	404	POINT (1873814.137 589052.339)
6036	06S02W34B006	Observation	Single Well	152	POINT (1861394.207 597916.983)



Tip

If we are interested in wells that fall within two or more counties (i.e., we have multiple records that will be used for selection), we can enclose the above code in a `for` loop.

[1] Clip Vector Data with GeoPandas, GeoPandas

[2] Indexing and Selecting Data, GeoPandas

[3] Indexing and selecting data, pandas

[4] How do I select a subset of a DataFrame?, pandas

[5] GeoSeries - Binary Predicates, GeoPandas

[6](1,2) geopandas.GeoSeries.within, GeoPandas

[7](1,2) Data Structures, GeoPandas

Learning Objectives

- Utilize different vector overlays and understand the differences between each
- Join data based on their geographic location and explore the different join types

Review

- Spatial Vector Data
- Attributes & Indexing for Vector Data
- Creating Spatial Vector Data

Spatial Overlays and Joins

Combining two or more datasets together is a fundamental aspect of GIS. Using `geopandas`, we can create new geometries from existing datasets by overlaying them on top of each other, identifying where they do and do not overlap, and deciding what parts we want to extract from these overlays. For each of these new shapes, the attribute data from the existing constituent datasets are also combined together. [1], [2]

In this chapter, we will focus on vector overlays, which involve combining vector data. We'll explore five types of vector overlays and merging: union, intersection, difference (erase), identity, and spatial join.

Setup

First, let's import the necessary modules (click the + below to show code cell).

```
# Import modules
import geopandas as gpd
import matplotlib.pyplot as plt
```

To illustrate these geoprocessing tools, we will utilize shapefiles of San Francisco Bay Area county boundaries, Bay Area watershed boundaries, and wells within the Bay Area and the surrounding 50 km. We will load in the data and reproject the data (click the + below to show code cell).

Important

All the data must have the same coordinate system in order for extraction to work correctly.

```
# Load data

# County boundaries
# Source: https://opendata.mtc.ca.gov/datasets/san-francisco-bay-region-counties-clipped?geometry=-125.590%2C37.123%2C-119.152%2C38.640
counties =
gpd.read_file("../_static/e_vector_shapefiles/sf_bay_counties/sf_bay_counties.shp"
)

# Watershed boundaries
# Source: https://gis.data.ca.gov/datasets/CDFW::epa-surf-your-watershed-ds732?geometry=-128.711%2C36.474%2C-115.835%2C39.504
# Modified by author so that only the watersheds with centroids in the Bay Area counties were kept
watersheds =
gpd.read_file("../_static/e_vector_shapefiles/sf_bay_watersheds/sf_bay_watersheds.shp")

# Well locations
# Source: https://gis.data.ca.gov/datasets/3a3e681b894644a9a95f9815aeeeb57f_0?geometry=-123.143%2C36.405%2C-119.230%2C37.175
# Modified by author so that only the well locations within the counties and the surrounding 50 km were kept
wells =
gpd.read_file("../_static/e_vector_shapefiles/sf_bay_wells_50km/sf_bay_wells_50km.shp")

# Reproject data to NAD83(HARN) / California Zone 3
# https://spatialreference.org/ref/epsg/2768/
proj = 2768
counties = counties.to_crs(proj)
watersheds = watersheds.to_crs(proj)
wells = wells.to_crs(proj)
```

We'll define some functions to make displaying and mapping our results a bit easier (click the + below to show code cell).

```
def display_table(table_name, attribute_table):
    '''Display the first and last two rows of attribute table.'''

    # Print title
    print("Attribute Table: {}".format(table_name))

    # Print number of rows and columns
    print("\nTable shape (rows, columns): {}".format(attribute_table.shape))

    # Display first two rows of attribute table
    print("\nFirst two rows:")
    display(attribute_table.head(2))

    # Display last two rows of attribute table
    print("\nLast two rows:")
    display(attribute_table.tail(2))


def plot_overlay(overlay_type, overlay_result):
    '''Plot the overlay result on a map and add the outlines of the original
shapefiles on top.'''

    # Create subplots
    fig, ax = plt.subplots(1, 1, figsize = (10, 10))

    # Plot overlay data
    overlay_result.plot(ax = ax, cmap = 'Set2', edgecolor = 'black')

    # Plot outlines of original shapefiles
    counties.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
    watersheds.plot(ax = ax, color = 'none', edgecolor = 'dodgerblue')

    # Stylize plots
    plt.style.use('bmh')

    # Set title
    ax.set_title('San Francisco Bay Area County and Watershed
Boundaries\n{}'.format(overlay_type), fontdict = {'fontsize': '15', 'fontweight' :
'3'})


def plot_merge(merge_type, merge_result, merge_vector):
    '''Plot the merge result on a map.'''

    # Create subplots
    fig, ax = plt.subplots(1, 1, figsize = (10, 10))

    # Plot data depending on vector type
    # For points
    if merge_vector == "point":

        # Plot data
        counties.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
        merge_result.plot(ax = ax, marker = 'o', color = 'dodgerblue', markersize
= 3)

    # For polygons
    else:

        # Plot data
        merge_result.plot(ax = ax, cmap = 'Set2', edgecolor = 'black')

    # Stylize plots
    plt.style.use('bmh')

    # Set title
    ax.set_title('San Francisco Bay Area County Boundaries and Well
Locations\n{}'.format(merge_type), fontdict = {'fontsize': '15', 'fontweight' :
'3'})
```

Overlays

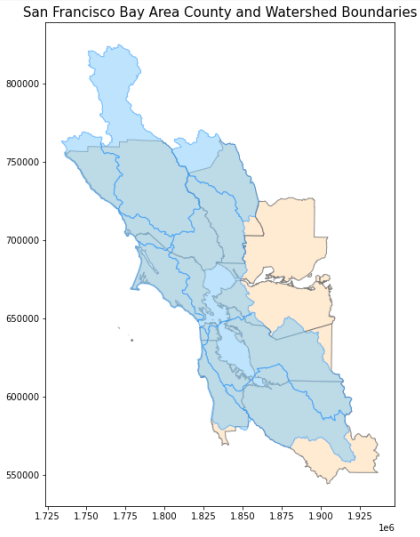
For the first four, we can use the `overlay` function in `geopandas`. We simply change the argument for the `how` parameter to the overlay of our choosing.

We will use the county boundaries and watershed boundaries shapefiles in these examples. The overlays will allow us to see what areas are only in a county, only in a watershed, or in both.

Let's briefly examine the attribute table of our shapefiles and plot the data so that we know what we're working with.

```
# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (10, 10))

# Plot data
counties.plot(ax = ax, color = 'bisque', edgecolor = 'dimgray', alpha = 0.75)
watersheds.plot(ax = ax, color = 'lightskyblue', edgecolor = 'dodgerblue', alpha =
0.55)

# Stylize plots
plt.style.use('bmh')

# Set title
ax.set_title('San Francisco Bay Area County and Watershed Boundaries', fontdict =
{'fontsize': '15', 'fontweight' : '3'})
```

```
Text(0.5, 1.0, 'San Francisco Bay Area County and Watershed Boundaries')
```




```
# Print attribute table
display(counties)
```

	coname	geometry
0	Alameda County	MULTIPOLYGON (((1860234.837 612219.122, 186007...
1	Contra Costa County	MULTIPOLYGON (((1836501.066 656512.837, 183649...
2	Marin County	MULTIPOLYGON (((1830493.060 653832.167, 183050...
3	Napa County	POLYGON ((1860171.549 724651.231, 1860170.736 ...
4	San Francisco County	MULTIPOLYGON (((1779231.793 635380.766, 177918...
5	San Mateo County	MULTIPOLYGON (((1836712.268 569216.860, 183671...
6	Santa Clara County	MULTIPOLYGON (((1858733.082 607586.450, 185874...
7	Solano County	MULTIPOLYGON (((1875491.908 673077.900, 187549...
8	Sonoma County	MULTIPOLYGON (((1746855.532 743026.706, 174685...

```
# Print attribute table
display(watersheds)
```

	CUNAME	geometry
0	RUSSIAN	POLYGON ((1772952.612 823099.656, 1775209.737 ...
1	UPPER_PUTAH	POLYGON ((1825517.283 769282.042, 1826065.366 ...
2	GUALALA-SALMON	POLYGON ((1747656.800 769294.908, 1748144.659 ...
3	SAN_PABLO_BAY	POLYGON ((1815758.957 741414.318, 1816340.883 ...
4	BODEGA_BAY	POLYGON ((1778048.116 714367.855, 1780284.096 ...
5	TOMALES-DRAKE_BAYS	POLYGON ((1783925.368 698933.179, 1784977.305 ...
6	SAN_FRANCISCO_BAY	POLYGON ((1856614.017 653898.716, 1856857.793 ...
7	SAN_FRANCISCO_COASTAL_SOUTH	POLYGON ((1826732.900 640369.438, 1827321.624 ...
8	COYOTE	POLYGON ((1875229.662 619208.108, 1875664.986 ...

Union

With `how='union'`, all data (all geometries regardless of overlap) is kept. [1]

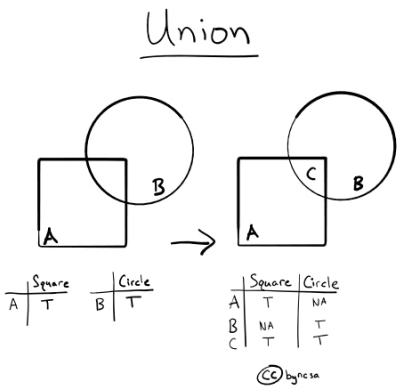


Fig. 39 Union keeps all the data. In the figure above, all of A and B are kept.

Looking at the attribute table, we see that the attributes from both individual datasets have been combined. The areas that are unique to one dataset (no overlap) have `NaN` as values in the fields that originated from the other dataset. [1], [2]

```
# Get union
union_result = gpd.overlay(counties, watersheds, how = 'union')

# Print head and tail of attribute table
display_table(table_name = "Union", attribute_table = union_result)
```

```
Attribute Table: Union

Table shape (rows, columns): (46, 3)

First two rows:
```

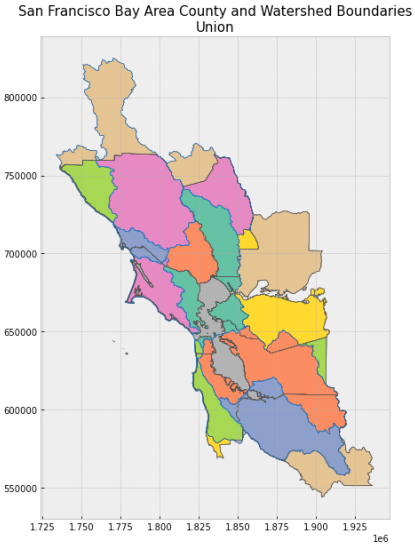
	coname	CUNAME	geometry
0	Alameda County	SAN_PABLO_BAY	POLYGON ((1844308.600 657338.949, 1844333.772 ...
1	Contra Costa County	SAN_PABLO_BAY	MULTIPOLYGON (((1850770.778 650276.858, 185076...

```
Last two rows:
```

	coname	CUNAME	geometry
44	None	SAN_FRANCISCO_COASTAL_SOUTH	MULTIPOLYGON (((1853276.563 580696.796, 185327...
45	None	COYOTE	MULTIPOLYGON (((1880333.343 567100.327, 188026...

Next, we can map the data, filling in the areas with color that have been retained. As the plot shows, no data was removed.

```
# Plot overlay
plot_overlay(overlay_type = "Union", overlay_result = union_result)
```



Intersection

With `how='intersection'`, only the areas where all datasets contain data (have geometries) are combined together. [1]

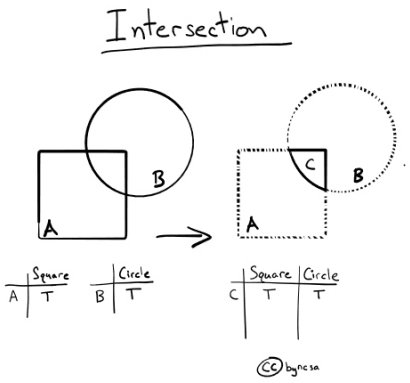


Fig. 40 Intersection keeps the geometries that overlap with each other. In the figure above, only the portion where A and B overlap is kept.

Because there are no areas unique to one dataset, notice how the attribute table of the combined dataset does not have any `NaN` values. When mapping the intersection overlay, we can see that any areas that did not have any overlay were discarded (areas with an outline but no fill). Areas covered by the county and watershed boundaries datasets are kept (shown in color). [1], [2]

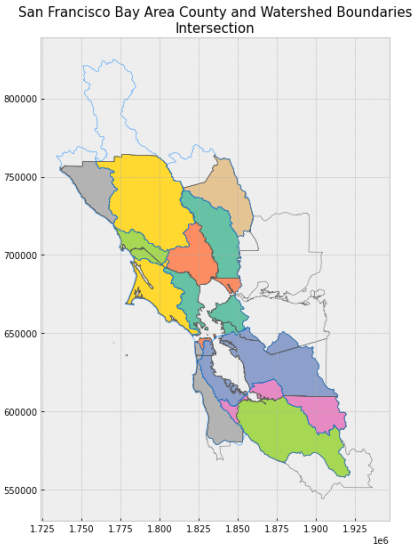
```
# Get intersection
intersection_result = gpd.overlay(counties, watersheds, how = 'intersection')

# Print head and tail of attribute table
display_table(table_name = "Intersection", attribute_table = intersection_result)

# Plot overlay
plot_overlay(overlay_type = "Intersection", overlay_result = intersection_result)
```

```
Attribute Table: Intersection

Table shape (rows, columns): (28, 3)

First two rows:
```

	coname	CUNAME	geometry
0	Alameda County	SAN_PABLO_BAY	POLYGON ((1844308.600 657338.949, 1844333.772 ...
1	Contra Costa County	SAN_PABLO_BAY	MULTIPOLYGON (((1850770.778 650276.858, 185076...

```
Last two rows:
```

	coname	CUNAME	geometry
26	Santa Clara County	SAN_FRANCISCO_COASTAL_SOUTH	MULTIPOLYGON (((1855641.060 585520.902, 185560...
27	Sonoma County	GUALALA-SALMON	MULTIPOLYGON (((1770306.496 719948.583, 177028...



Identity

With `how='identity'`, data from both layers are combined, but only the geometries that are unique to the first dataset or are covered by both datasets are kept. Any geometries unique to the second dataset (no overlapping with the first dataset) are discarded. [1]

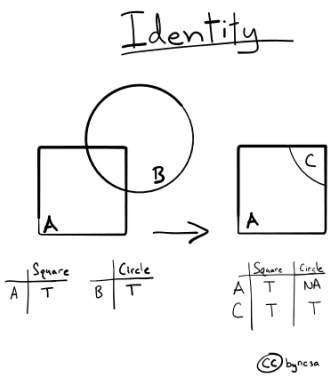


Fig. 41 Identity keeps the geometries of the first dataset. Any intersecting geometries from the second dataset are also combined and included. In the figure, all of A and the portion of B that intersects A are kept.

Looking at the attribute table, the fields from the individual datasets have been combined. For those geometries unique to the first dataset, the fields that came from the second dataset have `NaN` as values.

Looking at the map, we see all combined geometries except for the areas that are unique to the second dataset (watershed boundaries dataset).

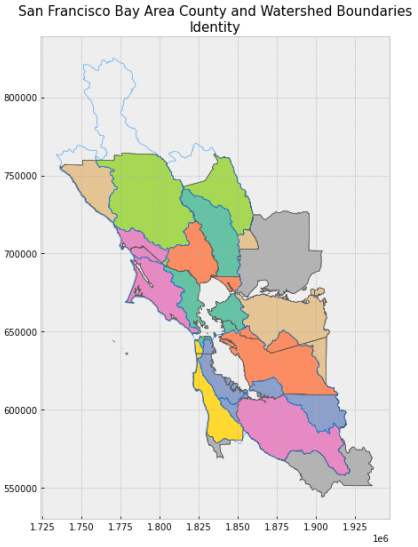
```
# Get identity
identity_result = gpd.overlay(counties, watersheds, how = 'identity')

# Print head and tail of attribute table
display_table(table_name = "Identity", attribute_table = identity_result)

# Plot overlay
plot_overlay(overlay_type = "Identity", overlay_result = identity_result)
```

```
Attribute Table: Identity

Table shape (rows, columns): (37, 3)

First two rows:
```

	coname	CUNAME	geometry
0	Alameda County	SAN_PABLO_BAY	POLYGON ((1844308.600 657338.949, 1844333.772 ...
1	Contra Costa County	SAN_PABLO_BAY	MULTIPOLYGON (((1850770.778 650276.858, 185076...

```
Last two rows:
```

	coname	CUNAME	geometry
35	Solano County	None	MULTIPOLYGON (((1893901.284 678662.321, 189390...
36	Sonoma County	None	MULTIPOLYGON (((1804439.333 758178.073, 180446...



Difference (Erase)

With `how='difference'`, the areas covered by the second dataset is used to "cut out" or erase those corresponding areas in the first dataset. In other words, only the areas in the first dataset that do not overlap with the second dataset are kept. [1], [2]

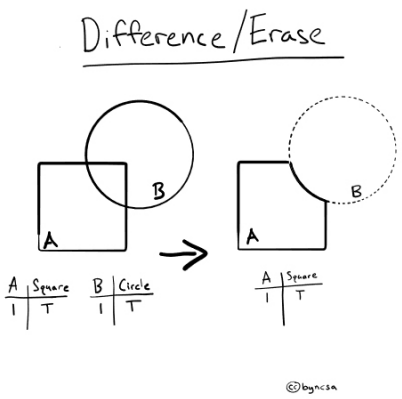


Fig. 42 Difference (erase) removes geometries that intersect with each other. In the figure above, B is used to cut out and remove a portion of A.

Looking at the attribute table, the fields from the second dataset do not appear in the combined dataset. The second dataset was "combined" with the first dataset by discarding some data (altering the geometry) from the first dataset.

Looking at the map, we only see areas of the first dataset (county dataset) that are not covered by the second dataset (watershed boundaries dataset).

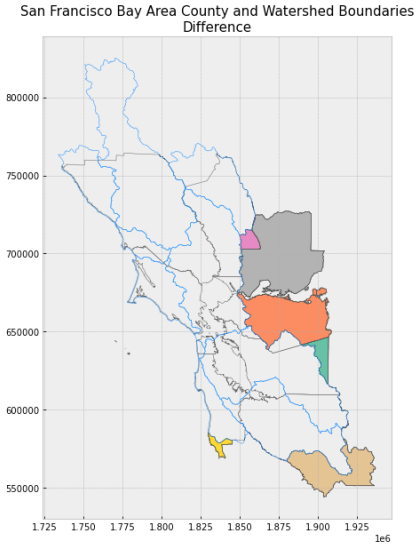
```
# Get difference
difference_result = gpd.overlay(counties, watersheds, how = 'difference')

# Print head and tail of attribute table
display_table(table_name = "Difference", attribute_table = difference_result)

# Plot overlay
plot_overlay(overlay_type = "Difference", overlay_result = difference_result)
```

```
Attribute Table: Difference

Table shape (rows, columns): (9, 2)

First two rows:
```

	coname	geometry
0	Alameda County	MULTIPOLYGON (((1913747.025 610819.209, 191379...
1	Contra Costa County	MULTIPOLYGON (((1866870.506 674116.607, 186699...

```
Last two rows:
```

	coname	geometry
7	Solano County	MULTIPOLYGON (((1879629.756 727080.818, 187969...
8	Sonoma County	MULTIPOLYGON (((1804439.333 758178.073, 180446...



Merge

Spatial Join

With spatial join, attributes from one dataset are appended to those in another dataset based on a specified relative spatial relationship. [3], [4]

In `geopandas`, we use the `sjoin()` function. In addition to passing the datasets as arguments, and we also pass arguments for two parameters `op` and `how`. The `op` parameter specifies the spatial relationship needed in order for the attributes of one feature to be joined to another. [3]

The following spatial relationships are available in `geopandas`:

Spatial Relationship	Description
`contains`	geometry's points are not to the exterior of the other geometry, provided that the geometry's interior contains at least one point of the other geometry's interior
`crosses`	geometry's interior intersects that of the other geometry, provided that the geometry does not contain the other and the dimension of the intersection is less than the dimension of either geometry
`intersects`	geometry's boundary or interior touches or crosses any part of the other geometry
`overlaps`	geometry shares at least one point, but not all points, with the other geometry, provided that the geometries and the intersection of their interiors all have the same dimensions
`touches`	geometry shares at least one point with the other geometry, provided that no part of the geometry's interior intersects with the other geometry
`within`	geometry is enclosed in the other geometry (geometry's boundary and interior intersects with the other geometry's interior only)

Note

These relationships are defined from the first dataset to the second dataset (for example, `contains` specifies that a feature from the first dataset must contain a feature from the second dataset for a join to occur).

Warning

Depending on the argument specified in the `op` parameter, a geometry that falls directly on the boundary of another geometry may be counted, may be counted twice, or may not be counted at all. For example, if a point falls on a boundary between two geometries, `op = "intersects"` will count that point twice and allocate (join) it to both geometries that share the boundary, whereas `op = "within"` will not count or allocate the point at all.

Just like regular table joins, there are multiple types of spatial joins, which determine which features from both datasets are kept in the output dataset. This is specified using the `how` parameter. [3], [4]

Join Type	Description
`left`	all features from the first or left dataset are kept, regardless if the feature met the specified spatial relationship criteria for a join/regardless if there is a match
`right`	all features from the second or right dataset are kept, regardless if the feature met the specified spatial relationship for a join
`inner`	only features from both datasets that met the spatial relationship and were joined are kept; the geometries from the first or left dataset are used for the join

We'll illustrate this geoprocessing using the county boundaries shapefile and the well locations shapefile. Let's quickly examine the wells attribute table and plot both datasets.

```
# Print head and tail of attribute table
display_table(table_name = "San Francisco Bay Area and Surrounding Area Wells",
attribute_table = wells)

# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (10, 10))

# Plot data
counties.plot(ax = ax, color = 'bisque', edgecolor = 'dimgray')
wells.plot(ax = ax, marker = 'o', color = 'dodgerblue', markersize = 3)

# Stylize plots
plt.style.use('bmh')

# Set title
ax.set_title('San Francisco Bay Area County Boundaries and Well Locations',
fontdict = {'fontsize': '15', 'fontweight' : '3'})
```

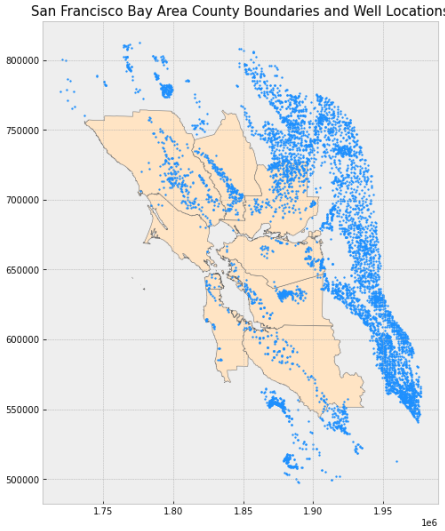
```
Attribute Table: San Francisco Bay Area and Surrounding Area Wells

Table shape (rows, columns): (6037, 5)

First two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
0	2400064-001	Public Supply	Single Well	0	POINT (1968290.923 592019.918)
1	2400099-001	Public Supply	Single Well	0	POINT (1969113.543 595876.691)

```
Last two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry
6035	None	Unknown	Unknown	0	POINT (1960933.648 610295.428)
6036	06S02W34B006	Observation	Single Well	152	POINT (1861394.207 597916.983)

```
Text(0.5, 1.0, 'San Francisco Bay Area County Boundaries and Well Locations')
```



Left Join

We'll first demonstrate a left join. Notice that all features from the left dataset (wells dataset) are kept. The features that did not meet the spatial relationship criteria for a join have `NaN` as values for the fields that originated from the right dataset (county boundaries dataset).

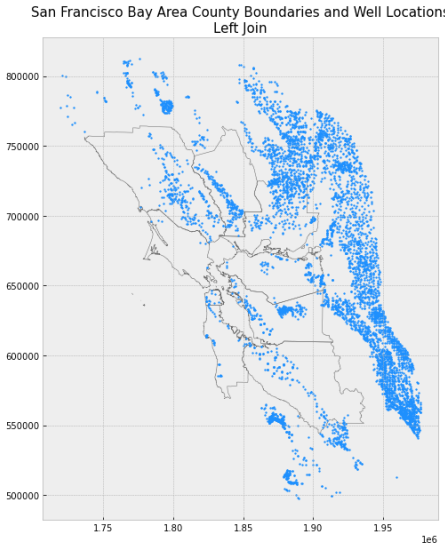
```
# Get inner join
left_join_result = gpd.sjoin(wells, counties, how = "left", op = 'within')

# Print head and tail of attribute table
display_table(table_name = "Left Join", attribute_table = left_join_result)

# Plot merge
plot_merge(merge_type = "Left Join", merge_result = left_join_result, merge_vector
= "point")
```

```
Attribute Table: Left Join

Table shape (rows, columns): (6037, 7)

First two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry	index_right	coname
0	2400064-001	Public Supply	Single Well	0	POINT (1968290.923 592019.918)	NaN	NaN
1	2400099-001	Public Supply	Single Well	0	POINT (1969113.543 595876.691)	NaN	NaN

```
Last two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry	index_right	coname
6035	None	Unknown	Unknown	0	POINT (1960933.648 610295.428)	NaN	NaN
6036	06S02W34B006	Observation	Single Well	152	POINT (1861394.207 597916.983)	6.0	Santa Clara County



Right Join

For a right join, all features from the right dataset (county boundaries dataset) are kept but are repeated multiple times. This is because a "new" county feature is created for every well point that falls within a county's boundary. As a result, because wells must fall within the county boundaries for a join to occur on the county boundaries feature, there are no resulting features with `NaN` as values in the attribute table.

> **Attention**
>
> The results here are a bit useless, since it's just each county boundary multiplied by the number of wells in that county, but we kept this example for comprehensiveness.

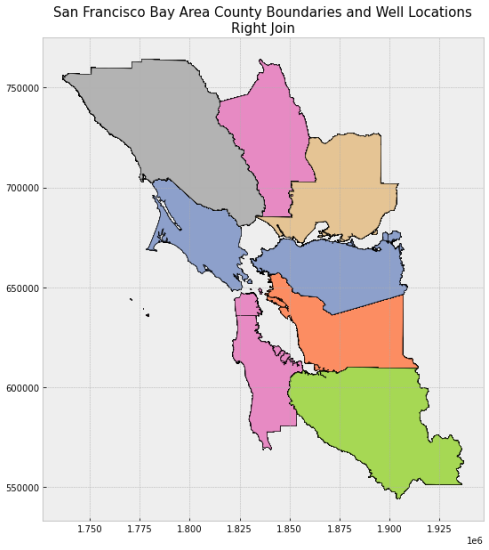
```
# Get inner join
right_join_result = gpd.sjoin(wells, counties, how = "right", op = 'within')

# Print head and tail of attribute table
display_table(table_name = "Right Join", attribute_table = right_join_result)

# Plot merge
plot_merge(merge_type = "Right Join", merge_result = right_join_result,
merge_vector = "polygon")
```

```
Attribute Table: Right Join

Table shape (rows, columns): (1337, 7)

First two rows:
```

	index_left	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	coname	geometry
0	787	3S/1E 9M 4	Industrial	Single Well	498	Alameda County	MULTIPOLYGON (((1860234.837 612219.122, 186007...
0	755	3S/1E 8H13	Observation	Single Well	800	Alameda County	MULTIPOLYGON (((1860234.837 612219.122, 186007...

```
Last two rows:
```

	index_left	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	coname	geometry
8	1697	AVCA-01	Residential	Single Well	100	Sonoma County	MULTIPOLYGON (((1746855.532 743026.706, 174685...
8	3508	11N10W08P001M	Residential	Single Well	30	Sonoma County	MULTIPOLYGON (((1746855.532 743026.706, 174685...



Inner Join

Finally, with an inner join, only the well locations that fall within the county boundaries are kept. These well locations have the county boundaries dataset appended to them. Because it's an inner join, there are no resulting features with `NaN` as values in the attribute table.

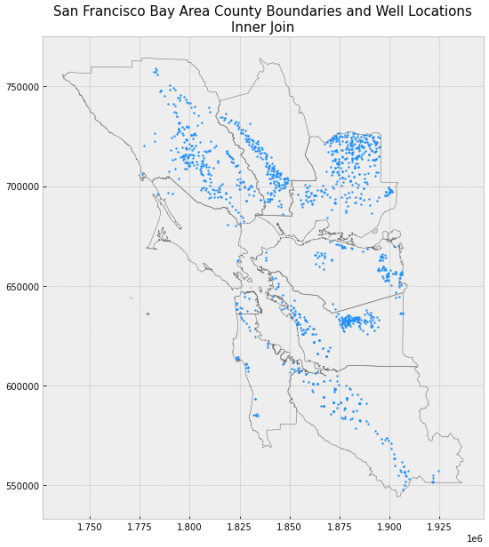
```
# Get inner join
inner_join_result = gpd.sjoin(wells, counties, how = "inner", op = 'within')

# Print head and tail of attribute table
display_table(table_name = "Inner Join", attribute_table = inner_join_result)

# Plot merge
plot_merge(merge_type = "Inner Join", merge_result = inner_join_result,
merge_vector = "point")
```

```
Attribute Table: Inner Join

Table shape (rows, columns): (1337, 7)

First two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry	index_right	coname
128	None	Stockwatering	Unknown	40	POINT (1882982.865 701605.988)	7	Solano County
129	None	Irrigation	Unknown	240	POINT (1884096.112 699303.331)	7	Solano County

```
Last two rows:
```

	WELL_NAME	WELL_USE	WELL_TYPE	WELL_DEPTH	geometry	index_right	coname
1759	T2S/R5W-06M01	Residential	Single Well	275	POINT (1827410.559 644648.215)	4	San Francisco County
2316	Kirkham MW130	Observation	Part of a nested/multi-completion well	130	POINT (1823285.838 641661.323)	4	San Francisco County



[1](1,2,3,4,5,6,7) Set-Operations with Overlay, GeoPandas

[2](1,2,3,4) GIS Fundamentals: A First Text on Geographic Information Systems, 5th ed., Paul Bolstad

[3](1,2,3) Merging Data, GeoPandas

[4](1,2) Spatial Join (Analysis), Esri

> **Learning Objectives**
>
> - Understand various types of spatial join relationships
> - Recognize different types of spatial joins
> - Understand how table relationships can affect a spatial join

Spatial Joins

To spatially join one dataset to another, attributes from the first dataset (join feature) are appended to the attributes in the second dataset (target feature) based on the relative spatial relationship between the two datasets' geometries. [1], [2]

> **Note**
>
> Unlike table joins by attributes, we're not really concerned with the tables having a primary key (a column, or columns, that uniquely identifies each record in a table) to conduct a join. Instead, the relative spatial location between the features in two datasets will determine what gets joined.

This chapter will cover spatial join relationships, spatial join types, and attribute table relationships.

Spatial Join Relationships

There are multiple spatial join relationships that we can specify. Only the features that meet the specified spatial relationship criteria will be joined together. [1], [2] The spatial arrangement necessary to satisfy a specified spatial relationship depends on the vector types (i.e., point, line, polygon) of the join feature and the target feature. Note that some spatial relationships are not possible for certain combinations of vector types.

Below is a description of the spatial relationships available in the `geopandas` module (this list is not necessarily exhaustive). [^gpd_meerge] We also provide diagrams visualizing the spatial arrangement needed for various combinations of vector types to satisfy the spatial relationship criteria and allow for a join.

Contains

One feature contains another if the geometry's points are not to the exterior of the other's geometry and the geometry's interior contains at least one point of the other geometry's interior.

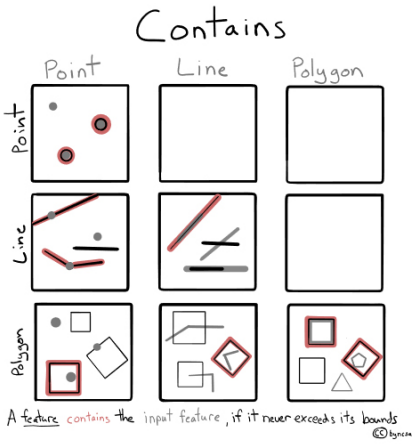


Fig. 43 Spatial arrangements for *contains* for different combinations of vector types.

Crosses

The geometry's interior *crosses* that of the other geometry, provided that the geometry does not contain the other and the dimension of the intersection is less than the dimension of either geometry.

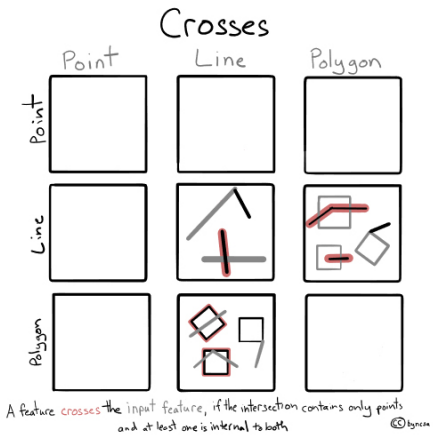


Fig. 44 Spatial arrangements for *crosses* for different combinations of vector types.

Intersects

A feature *intersects* another if the geometry's boundary or interior has any part in common with the other geometry.

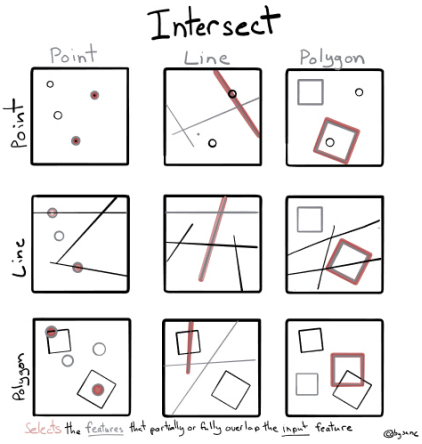


Fig. 45 Spatial arrangements for *intersects* for different combinations of vector types.

Overlaps

A feature *overlaps* another if the geometry shares at least one point, but not all points, with the other geometry, provided that the geometries and the intersection of their interiors all have the same dimensions.

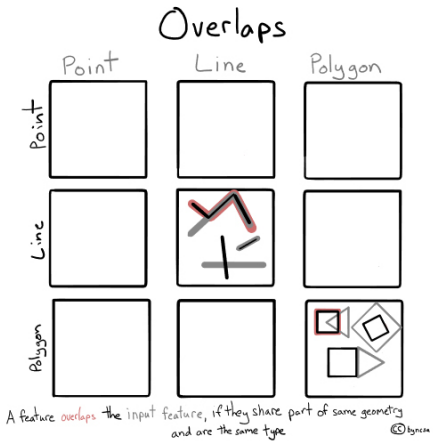


Fig. 46 Spatial arrangements for *overlaps* for different combinations of vector types.

Touches

A feature *touches* another if the geometry shares at least one point with the other geometry, provided that no part of the geometry's interior intersects with the other geometry.

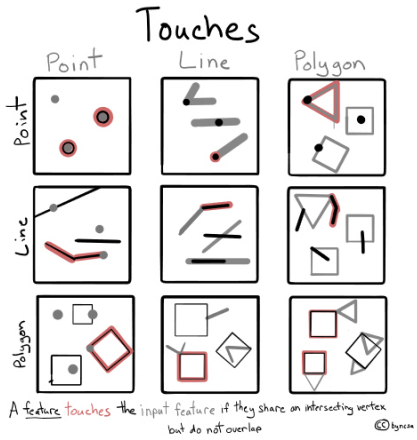


Fig. 47 Spatial arrangements for *touches* for different combinations of vector types.

Within

A feature is *within* another if the geometry is enclosed in the other geometry (geometry's boundary and interior intersects with the other geometry's interior only).

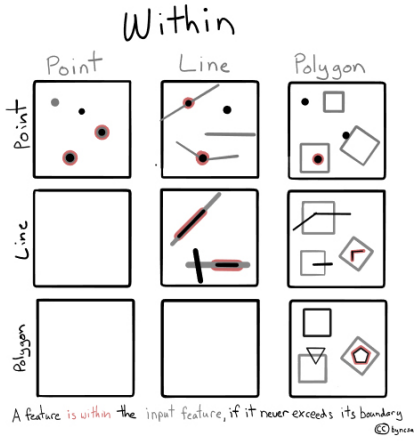


Fig. 48 Spatial arrangements for *within* for different combinations of vector types.

Spatial Join Types

There are four types of spatial joins: outer join, inner join, left join, and right join. These spatial join types determine which features from both datasets are kept in the resulting output dataset.

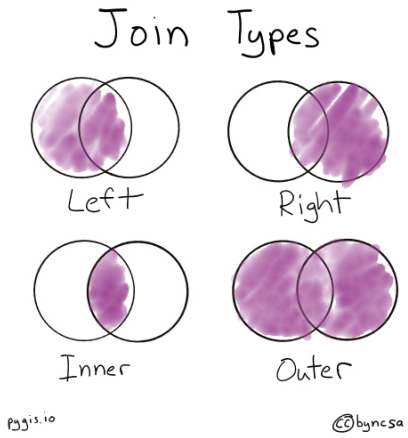


Fig. 49 There are four types of spatial joins. These Venn diagrams depict which features from both datasets are kept when they are joined together for each join type.

Only the inner, left, and right join types are available in the `geopandas` module and are identical to those in `pandas`. [2]

Outer join

All features from both datasets are kept, regardless if the features meet the specified spatial relationship criteria for a join. As all attribute fields are combined, rows that do not have a match may have null values in the fields that originated from the other dataset. [3]

Inner join

Only features from both datasets that meet the spatial relationship for the joined are kept. The geometries from the first or left dataset are used for the join. [3]

Left join

All features from the first or left dataset are kept, regardless if the features meet the specified spatial relationship criteria for a join. As all attribute fields are combined, rows that do not have a match with the right dataset may have null values in the fields that originated from the right dataset.

Right join

All features from the second or right dataset are kept, regardless if the features meet the specified spatial relationship criteria for a join. As all attribute fields are combined, rows that do not have a match with the left dataset may have null values in the fields that originated from the left dataset.

For more details on join types, see the chapter on joining by attributes.

Table Relationships

Table relationships, or cardinality, specify the direction of the join and the relationship between the two datasets. There are four types of table relationships: one-to-one, one-to-many, many-to-one, and many-to-many. While these relationships are not specified as a parameter in the `geopandas` module when conducting a spatial join, it is important to have a general idea of the table relationship between the two datasets. Generally, a many-to-one join and a many-to-many join should be avoided because it can create ambiguity and unpredictability when analyzing and manipulating the resulting attribute table. [3]

One-to-one relationship

Each feature in one geometry is linked to one (and only one) feature in the second geometry (and vice versa). Each feature in both datasets will appear only once at most. [3]

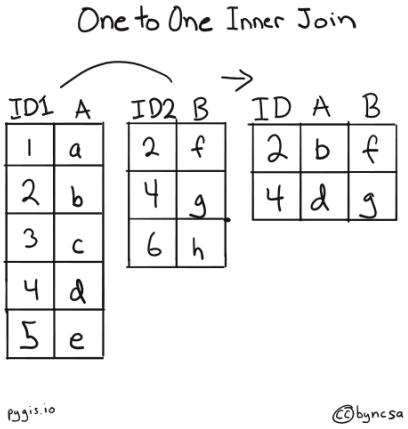


Fig. 50 Two joined tables have a one-to-one relationship when each feature in one geometry is linked to one (and only one) feature in the second geometry.

One-to-many relationship or many-to-one relationship

Each feature in one geometry is linked to one or more features in the second geometry. The output geometry may see the “one” feature duplicated across multiple rows to accommodate its multiple linkages in the other dataset (the “many”). A one-to-many or many-to-one relationship can be dependent on the spatial join type (for example, if a left join results in a one-to-many relationship, then a right join will result in a many-to-one relationship). [3]

Warning

These two table relationships—while similar—are not the same! In a one-to-many relationship, the “many” geometry is the dataset that is being joined to. Alternatively, in a many-to-one relationship, the “one” geometry is the dataset being joined to. Thus, the many-to-one relationship can result in duplicate “one” geometries and create uncertainty, which is why a many-to-one join should be avoided. [3]

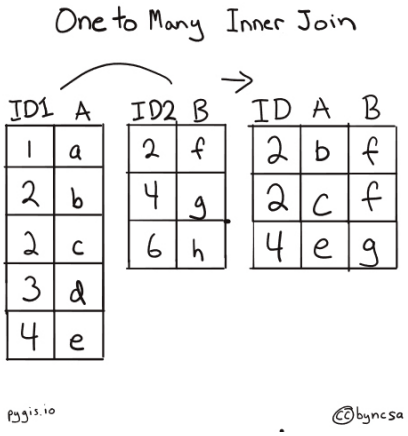


Fig. 51 Two joined tables have a one-to-many relationship when each feature in one geometry is linked to one or more features in the second geometry.

Many-to-many relationship

One or more features in one geometry are linked to one or more features in the second geometry. The output geometry may have multiple rows of each target and join feature to accommodate the multiple linkages, which can create ambiguity—hence why a many-to-many join should be avoided. [3]

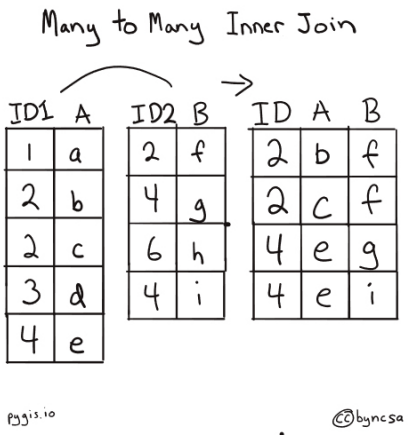


Fig. 52 Two joined tables have a many-to-many relationship when one or more features in one geometry are linked to one or more features in the second geometry.

Tip

To ensure you’re doing a one-to-one or one-to-many join, it may be helpful to envision your spatial join before performing a spatial join. Think about what geometries you will be joining to (the target feature) and what geometry or geometries from the join feature might be joined to one geometry in the target feature.

---

[1](1,2) Spatial Join (Analysis), Esri

[2](1,2,3) Merging Data, GeoPandas

[3](1,2,3,4,5,6,7) GIS Fundamentals: A First Text on Geographic Information Systems, 5th ed., Paul Bolstad

---

Learning Objectives

- Create a grid to bin features
- Bin features using the grid
- Display kernel density estimation results and export resulting raster

Review

- Spatial Vector Data
- Spatial Raster Data
- Attributes & Indexing for Vector Data
- Creating Spatial Vector Data

Point Density Measures - Counts & Kernel Density

Summary operations are useful for aggregating data, whether it be for analyzing overall trends or visualizing concentrations of data. Summarizing allows for effective analysis and communication of the data as compared to simply looking at or displaying points, lines, and polygons on a map.

This chapter will explore two summary operations that highlight concentrations of data: count points in a rectangular or hexagonal grid or by polygon and kernel density.

Setup

First, we will import the necessary modules (click the + below to show code cell).

```
# Import modules
import geopandas as gpd
import geoplot as gplt
import math
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import rasterio
from rasterio.transform import Affine
from scipy import stats
from shapely.geometry import Polygon, box
from sklearn.datasets import fetch_species_distributions
from sklearn.neighbors import KernelDensity
```

We will utilize shapefiles of San Francisco Bay Area county boundaries and wells within the Bay Area and the surrounding 50 km. We will load in the data and reproject the data (click the + below to show code cell).

```
# Load data

# County boundaries
# Source: https://opendata.mtc.ca.gov/datasets/san-francisco-bay-region-counties-clipped?geometry=-125.590%2C37.123%2C-119.152%2C38.640
counties =
gpd.read_file("../_static/e_vector_shapefiles/sf_bay_counties/sf_bay_counties.shp"
)

# Well locations
# Source: https://gis.data.ca.gov/datasets/3a3e681b894644a9a95f9815aeeeb57f_0?geometry=-123.143%2C36.405%2C-119.230%2C37.175
# Modified by author so that only the well locations within the counties and the surrounding 50 km were kept
wells =
gpd.read_file("../_static/e_vector_shapefiles/sf_bay_wells_50km/sf_bay_wells_50km.shp")

# Reproject data to NAD83(HARN) / California Zone 3
# https://spatialreference.org/ref/epsg/2768/
proj = 2768
counties = counties.to_crs(proj)
wells = wells.to_crs(proj)

# Create a column that assigns each well a number
wells["Well_ID"] = np.arange(wells.shape[0])
```

Count Points in Rectangular or Hexagonal Grid or by Polygon

To summarize by grid, we create a new polygon layer consisting of a grid and overlay on another feature. We can summarize an aspect of that feature within each cell of the grid. The polygon layer commonly consists of a fishnet (rectangular cells), but using hexagons as a grid is becoming increasingly widespread.

Let's define a function that will create a grid of either rectangles or hexagons of a specified side length.

```
def create_grid(feature, shape, side_length):
    '''Create a grid consisting of either rectangles or hexagons with a specified
side length that covers the extent of input feature.'''

    # Slightly displace the minimum and maximum values of the feature extent by
creating a buffer
    # This decreases likelihood that a feature will fall directly on a cell
boundary (in between two cells)
    # Buffer is projection dependent (due to units)
    feature = feature.buffer(20)

    # Get extent of buffered input feature
    min_x, min_y, max_x, max_y = feature.total_bounds


    # Create empty list to hold individual cells that will make up the grid
    cells_list = []

    # Create grid of squares if specified
    if shape in ["square", "rectangle", "box"]:

        # Adapted from https://james-
brennan.github.io/posts/fast_gridding_geopandas/
        # Create and iterate through list of x values that will define column
positions with specified side length
        for x in np.arange(min_x - side_length, max_x + side_length, side_length):

            # Create and iterate through list of y values that will define row
positions with specified side length
            for y in np.arange(min_y - side_length, max_y + side_length,
side_length):

                # Create a box with specified side length and append to list
                cells_list.append(box(x, y, x + side_length, y + side_length))


    # Otherwise, create grid of hexagons
    elif shape == "hexagon":

        # Set horizontal displacement that will define column positions with
specified side length (based on normal hexagon)
        x_step = 1.5 * side_length

        # Set vertical displacement that will define row positions with specified
side length (based on normal hexagon)
        # This is the distance between the centers of two hexagons stacked on top
of each other (vertically)
        y_step = math.sqrt(3) * side_length

        # Get apothem (distance between center and midpoint of a side, based on
normal hexagon)
        apothem = (math.sqrt(3) * side_length / 2)

        # Set column number
        column_number = 0

        # Create and iterate through list of x values that will define column
positions with vertical displacement
        for x in np.arange(min_x, max_x + x_step, x_step):

            # Create and iterate through list of y values that will define column
positions with horizontal displacement
            for y in np.arange(min_y, max_y + y_step, y_step):

                # Create hexagon with specified side length
                hexagon = [[x + math.cos(math.radians(angle)) * side_length, y +
math.sin(math.radians(angle)) * side_length] for angle in range(0, 360, 60)]

                # Append hexagon to list
                cells_list.append(Polygon(hexagon))

            # Check if column number is even
            if column_number % 2 == 0:

                # If even, expand minimum and maximum y values by apothem value to
vertically displace next row
                # Expand values so as to not miss any features near the feature
extent
                min_y -= apothem
                max_y += apothem

            # Else, odd
            else:

                # Revert minimum and maximum y values back to original
                min_y += apothem
                max_y -= apothem

            # Increase column number by 1
            column_number += 1

    # Else, raise error
    else:
        raise Exception("Specify a rectangle or hexagon as the grid shape.")

    # Create grid from list of cells
    grid = gpd.GeoDataFrame(cells_list, columns = ['geometry'], crs = proj)

    # Create a column that assigns each grid a number
    grid["Grid_ID"] = np.arange(len(grid))

    # Return grid
    return grid
```

We will illustrate this methodology by counting the number of well points within each cell of the grid. There are two different ways we can accomplish this methodology, both with advantages and disadvantages.

To begin, we will set some global parameters for both examples.

```
# Set side length for cells in grid
# This is dependent on projection chosen as length is in units specified in
projection
side_length = 5000

# Set shape of grid
shape = "hexagon"
# shape = "rectangle"
```

Method 1 - Group by

This method involves using spatial joins to allocate each point to the cell in which it resides. All the points within each cell are subsequently grouped together and counted.

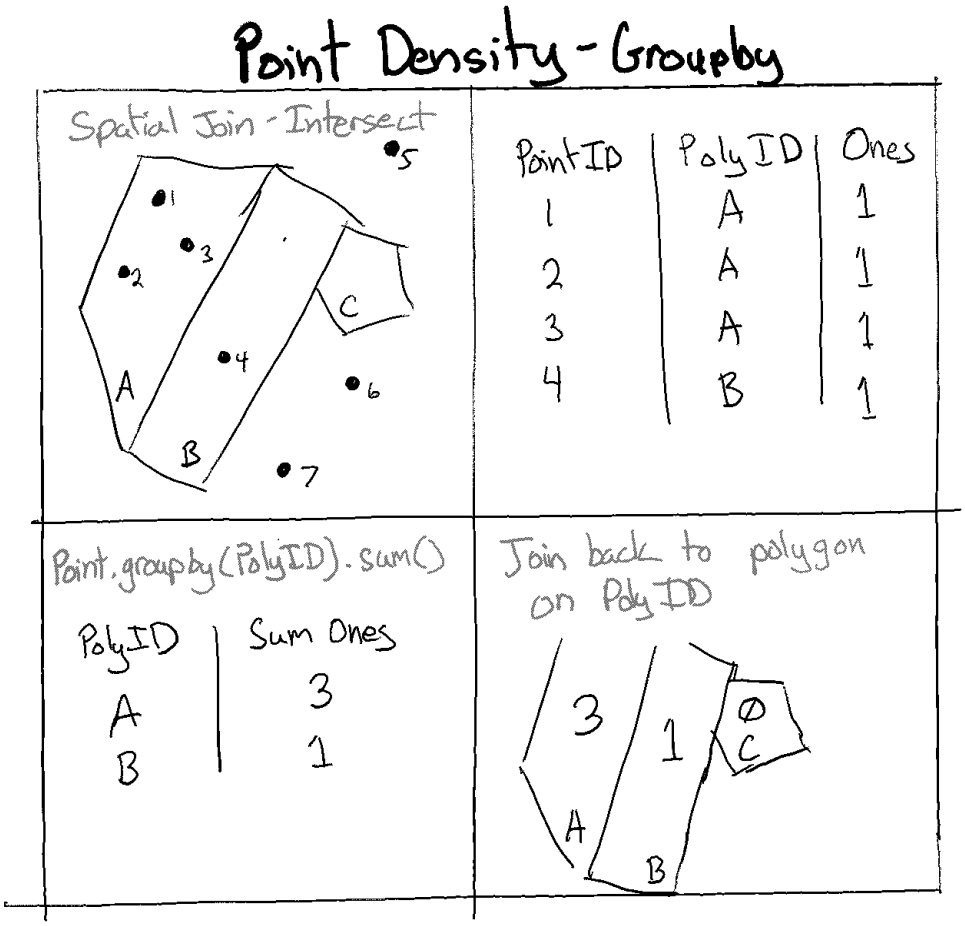
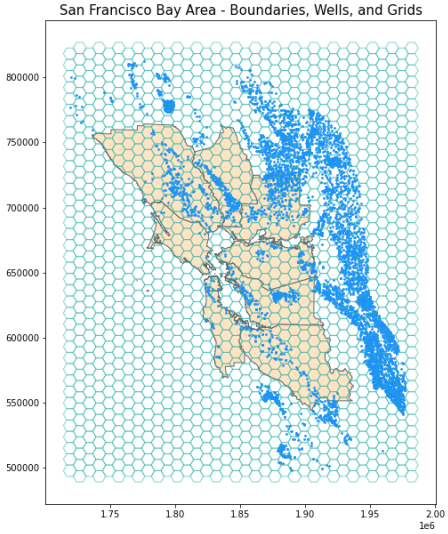


Fig. 53 Point density using groupby and geopandas

First, we will create a grid over the Bay Area.

```
# Create grid
bay_area_grid = create_grid(feature = wells, shape = shape, side_length =
side_length)

# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (10, 10))

# Plot data
counties.plot(ax = ax, color = 'bisque', edgecolor = 'dimgray')
wells.plot(ax = ax, marker = 'o', color = 'dodgerblue', markersize = 3)
bay_area_grid.plot(ax = ax, color = 'none', edgecolor = 'lightseagreen', alpha =
0.55)

# Set title
ax.set_title('San Francisco Bay Area - Boundaries, Wells, and Grids', fontdict =
{'fontsize': '15', 'fontweight' : '3'})
```

```
Text(0.5, 1.0, 'San Francisco Bay Area - Boundaries, Wells, and Grids')
```



Next, we will conduct a spatial join for each well point, essentially assigning it to a cell. We can add a field with a value of `1`, group all the wells in a cell, and aggregate **(sum)** all those `1` values to get the total number of wells in a cell.

```
# Perform spatial join, merging attribute table of wells point and that of the
cell with which it intersects
# op = "intersects" also counts those that fall on a cell boundary (between two
cells)
# op = "within" will not count those fall on a cell boundary
wells_cell = gpd.sjoin(wells, bay_area_grid, how = "inner", op = "intersects")

# Remove duplicate counts
# With intersect, those that fall on a boundary will be allocated to all cells
that share that boundary
wells_cell = wells_cell.drop_duplicates(subset = ['Well_ID']).reset_index(drop =
True)

# Set field name to hold count value
count_field = "Count"

# Add a field with constant value of 1
wells_cell[count_field] = 1

# Group GeoDataFrame by cell while aggregating the Count values
wells_cell = wells_cell.groupby('Grid_ID').agg({count_field:'sum'})

# Merge the resulting grouped dataframe with the grid GeoDataFrame, using a left
join to keep all cell polygons
bay_area_grid = bay_area_grid.merge(wells_cell, on = 'Grid_ID', how = "left")

# Fill the NaN values (cells without any points) with 0
bay_area_grid[count_field] = bay_area_grid[count_field].fillna(0)

# Convert Count field to integer
bay_area_grid[count_field] = bay_area_grid[count_field].astype(int)

# Display grid attribute table
display(bay_area_grid)
```

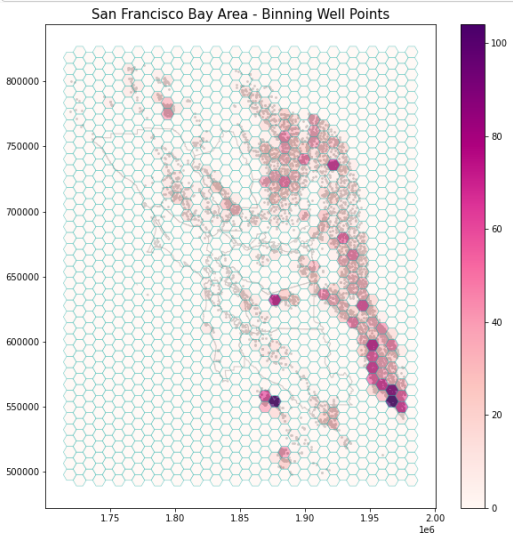
	geometry	Grid_ID	Count
0	POLYGON ((1724272.398 497763.821, 1721772.398 ...	0	0
1	POLYGON ((1724272.398 506424.075, 1721772.398 ...	1	0
2	POLYGON ((1724272.398 515084.329, 1721772.398 ...	2	0
3	POLYGON ((1724272.398 523744.583, 1721772.398 ...	3	0
4	POLYGON ((1724272.398 532404.837, 1721772.398 ...	4	0
...	...	...	...
1381	POLYGON ((1986772.398 787882.331, 1984272.398 ...	1381	0
1382	POLYGON ((1986772.398 796542.585, 1984272.398 ...	1382	0
1383	POLYGON ((1986772.398 805202.839, 1984272.398 ...	1383	0
1384	POLYGON ((1986772.398 813863.093, 1984272.398 ...	1384	0
1385	POLYGON ((1986772.398 822523.347, 1984272.398 ...	1385	0

1386 rows × 3 columns

We can plot the data to see how it looks.

```
# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (10, 10))

# Plot data
counties.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
wells.plot(ax = ax, marker = 'o', color = 'dimgray', markersize = 3)
bay_area_grid.plot(ax = ax, column = "Count", cmap = "RdPu", edgecolor =
'lightseagreen', linewidth = 0.5, alpha = 0.70, legend = True)

# Set title
ax.set_title('San Francisco Bay Area - Binning Well Points', fontdict =
{'fontsize': '15', 'fontweight' : '3'})
```

```
Text(0.5, 1.0, 'San Francisco Bay Area - Binning Well Points')
```



The advantage of this method is that it is pretty fast. To verify that all points have been counted once, we can check the aggregate of all the point sums for each cell.

```
# Check total number of well points counted and compare to number of well points
in input data
print("Total number of well points counted: {}\nNumber of well points in input
data: {}".format(sum(bay_area_grid.Count), len(wells)))
```

```
Total number of well points counted: 6037
Number of well points in input data: 6037
```

Method 2 - Iterate through each feature

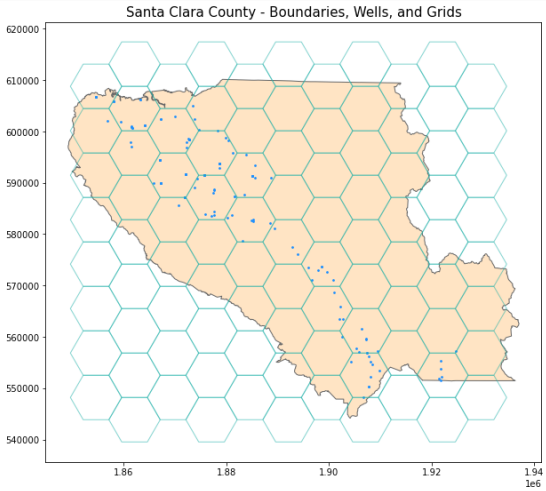
This second method is slightly more intuitive, but it can take a long time to run. We will use a subset of the input data–those that fall within Santa Clara County–to illustrate this example. We will first subset our data to Santa Clara County (click the + below to show code cell).

```
# Select the Santa Clara County boundary
sc_county = counties[counties["coname"] == "Santa Clara County"]

# Subset the GeoDataFrame by checking which wells are within Santa Clara County
sc_county_wells = wells[wells.within(sc_county.geometry.values[0])]
```

Next, we will create a grid over Santa Clara County.

```
# Create grid
sc_county_grid = create_grid(feature = sc_county_wells, shape = shape, side_length
= side_length)

# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (10, 10))

# Plot data
sc_county.plot(ax = ax, color = 'bisque', edgecolor = 'dimgray')
sc_county_wells.plot(ax = ax, marker = 'o', color = 'dodgerblue', markersize = 3)
sc_county_grid.plot(ax = ax, color = 'none', edgecolor = 'lightseagreen', alpha =
0.55)

# Set title
ax.set_title('Santa Clara County - Boundaries, Wells, and Grids', fontdict =
{'fontsize': '15', 'fontweight' : '3'})
```

```
Text(0.5, 1.0, 'Santa Clara County - Boundaries, Wells, and Grids')
```



We iterate through each cell in the grid and set a counter for each cell. We iterate through each well point and see if it is within (or intersects) the cell. If it is, the counter is increased by 1, and the feature is “discarded” so that it won’t be counted again (resolving the issue of a point falling on the boundary between two cells).

```
# Create empty list used to hold count values for each grid
counts_list = []

# Create empty list to hold index of points that have already been matched to a
grid
counted_points = []

# Iterate through each cell in grid
for i_1 in range(0, sc_county_grid.shape[0]):

    # Get a cell by index
    cell = sc_county_grid.iloc[[i_1]]

    # Reset index of cell to 0
    cell = cell.reset_index(drop = True)

    # Set point count to 0
    count = 0

    # Iterate through each feature in wells dataset
    for i_2 in range(0, sc_county_wells.shape[0]):

        # Check if index of point is in list of indices whose points have already
been matched to a grid and counted
        if i_2 in counted_points:

            # If already counted, skip remaining statements in loop and start at
top of loop
            continue

        # Otherwise, continue with remaining statements
        else:
            pass

        # Get a well point by index
        well = sc_county_wells.iloc[[i_2]]

        # Reset index of well point (to 0) to match the index-reset cell
        well = well.reset_index(drop = True)

        # Check if well intersects the cell
        # Best to use intersects instead of within or contains, as intersect will
count points that fall exactly on the cell boundaries
        # Points that fall exactly on a cell boundary (between two cells) will be
allocated to the first of the two cells called in script
        criteria_met = well.intersects(cell)[0]

        # If preferred, can check if well is within cell or if cell contains well
        # Both statements do the same thing
        # criteria_met = well.within(cell)[0]
        # criteria_met = cell.contains(well)[0]

        # Check if criteria has been met (True)
        if criteria_met:

            # If True, increase counter by 1 for the cell
            count += 1

            # Add index of counted point to the list
            counted_points.append(i_2)

        # Otherwise, criteria is not met (False)
        else:
            pass

    # Add total count for that cell to the list of counts
    counts_list.append(count)

# print(counts_list)

# Add a new column to the grid GeoDataFrame with the list of counts for each cell
sc_county_grid['Count'] = pd.Series(counts_list)

# Display grid attribute table
display(sc_county_grid)
```

	geometry	Grid_ID	Count
0	POLYGON ((1859618.426 548178.942, 1857118.426 ...	0	0
1	POLYGON ((1859618.426 556839.196, 1857118.426 ...	1	0
2	POLYGON ((1859618.426 565499.450, 1857118.426 ...	2	0
3	POLYGON ((1859618.426 574159.704, 1857118.426 ...	3	0
4	POLYGON ((1859618.426 582819.958, 1857118.426 ...	4	0
...	...	...	...
88	POLYGON ((1934618.426 574159.704, 1932118.426 ...	88	0
89	POLYGON ((1934618.426 582819.958, 1932118.426 ...	89	0
90	POLYGON ((1934618.426 591480.212, 1932118.426 ...	90	0
91	POLYGON ((1934618.426 600140.466, 1932118.426 ...	91	0
92	POLYGON ((1934618.426 608800.720, 1932118.426 ...	92	0

93 rows × 3 columns

We can check to make sure all well points in Santa Clara County were counted.

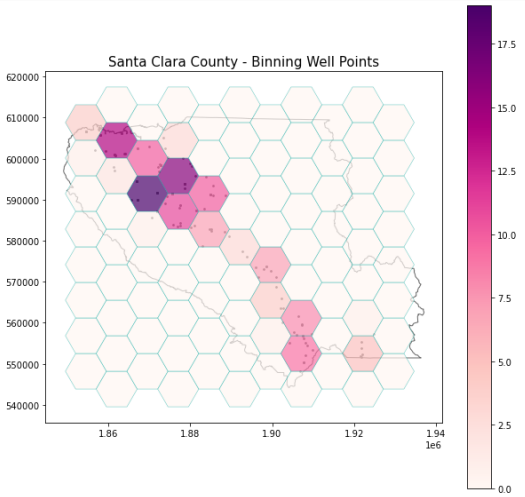
```
# Check total number of well points counted and compare to number of well points
in input data
print("Total number of well points counted: {}\nNumber of well points in input
data: {}".format(sum(counts_list), len(sc_county_wells)))
```

```
Total number of well points counted: 136
Number of well points in input data: 136
```

Finally, we can plot the data.

```
# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (10, 10))

# Plot data
sc_county.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
sc_county_wells.plot(ax = ax, marker = 'o', color = 'dimgray', markersize = 3)
sc_county_grid.plot(ax = ax, column = "Count", cmap = "RdPu", edgecolor =
'lightseagreen', linewidth = 0.5, alpha = 0.70, legend = True)

# Set title
ax.set_title('Santa Clara County - Binning Well Points', fontdict = {'fontsize':
'15', 'fontweight' : '3'})
```

```
Text(0.5, 1.0, 'Santa Clara County - Binning Well Points')
```



Kernel Density Estimation

Kernel density estimation (KDE) visualizes concentrations points or polylines. It calculates a magnitude per unit area, providing the density estimate of features within a specified neighborhood surrounding each feature. [1], [2]

A kernel function is used to fit a smooth surface to each feature. One of the most common types of kernels is the Gaussian kernel, which is a normal density function. Other types of kernel functions can be used, and the type affects the influence of surrounding points on a location's density estimate as the points' distances increase from that location. These kernels' functions vary in shapes and characteristics, such as where the function peaks, how pointed the peak is, and how fast the peak is reached with distance. [1], [2]

In addition to specifying a kernel, the bandwith can also be specified. This parameter defines how spread out the kernel is. A lower bandwith allows points far away from a location to affect the density estimate at that location, whereas with a higher bandwith, only close points have influence. [1], [2]

Individual density functions based on a specified kernel are plotted for each feature. Then, individual density function values at a location are aggregated to produce the KDE value at that location, and this is repeated across the entire point or polyline extent. The final KDE result is a raster depicting the sum of all individual density functions. [1], [2]

For more information on KDE, check out this visualization.

We will demonstrate two ways to perform kernel density estimation. The first way allows us to quickly visualize the KDE. The second way also allows us to export and save a KDE raster for additional analysis.

> **Tip**
>
> We are intentionally keeping the well points beyond (but within 50 km) of the Bay Area boundaries. This provides a buffer to ensure that the KDE for wells data near the boundaries is not inadvertently influenced by these artificial county boundaries. Once KDE is run, the result can be clipped to the Bay Area boundaries (which we do in the first method).

Method 1 - Display

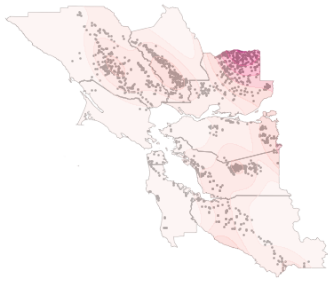
This method uses `geoplot`, a high-level plotting library for spatial data that complements `matplotlib`. For more information on `geoplot`, check out the documentation.

```
# Set projection to WGS 84 and reproject data
proj_wgs = 4326
counties_wgs = counties.to_crs(proj_wgs)
wells_wgs = wells.to_crs(proj_wgs)

# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (10, 10))

# Plot data
counties_wgs.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
wells_wgs.plot(ax = ax, marker = 'o', color = 'dimgray', markersize = 3)
gplt.kdeplot(wells_wgs, cmap = "RdPu", shade = True, clip = counties_wgs, thresh =
0, ax = ax, alpha = 0.5)

# Set title
ax.set_title('San Francisco Bay Area - Kernel Density Estimation for Wells',
fontdict = {'fontsize': '15', 'fontweight' : '3'})
```

```
/home/mmann1123/anaconda3/envs/pygisbookgw/lib/python3.7/site-
packages/seaborn/_decorators.py:43: FutureWarning: Pass the following variable as
a keyword arg: y. From version 0.12, the only valid positional argument will be
`data`, and passing other arguments without an explicit keyword will result in an
error or misinterpretation.
  FutureWarning
/home/mmann1123/anaconda3/envs/pygisbookgw/lib/python3.7/site-
packages/seaborn/distributions.py:1676: UserWarning: `shade_lowest` is now
deprecated in favor of `thresh`. Setting `thresh=0`, but please update your code.
  warnings.warn(msg, UserWarning)
```

```
Text(0.5, 1.0, 'San Francisco Bay Area - Kernel Density Estimation for Wells')
```

San Francisco Bay Area - Kernel Density Estimation for Wells



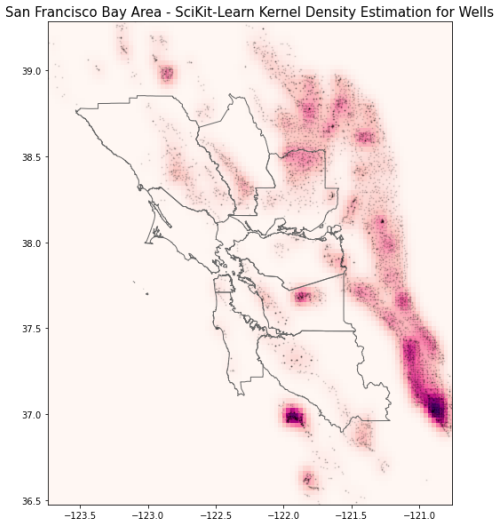
Method 2 - Display and export with `scikit-learn`

This method uses `scikit-learn` to visualize and export the KDE result. We are able to specify and change various estimator parameters. Examples include:

- kernel type
- metric (how distances from a location are calculated)
- algorithm (how to quickly identify neighboring points instead of performing time and resource intensive brute force)

For further reading, check out the **`scikit-learn`** documentation and the associated example.

```
# Get X and Y coordinates of well points
x_sk = wells_wgs["geometry"].x
y_sk = wells_wgs["geometry"].y

# Get minimum and maximum coordinate values of well points
min_x_sk, min_y_sk, max_x_sk, max_y_sk = wells_wgs.total_bounds

# Create a cell mesh grid
# Horizontal and vertical cell counts should be the same
XX_sk, YY_sk = np.mgrid[min_x_sk:max_x_sk:100j, min_y_sk:max_y_sk:100j]

# Create 2-D array of the coordinates (paired) of each cell in the mesh grid
positions_sk = np.vstack([XX_sk.ravel(), YY_sk.ravel()]).T

# Create 2-D array of the coordinate values of the well points
Xtrain_sk = np.vstack([x_sk, y_sk]).T

# Get kernel density estimator (can change parameters as desired)
kde_sk = KernelDensity(bandwidth = 0.04, metric = 'euclidean', kernel =
'gaussian', algorithm = 'auto')

# Fit kernel density estimator to wells coordinates
kde_sk.fit(Xtrain_sk)

# Evaluate the estimator on coordinate pairs
Z_sk = np.exp(kde_sk.score_samples(positions_sk))

# Reshape the data to fit mesh grid
Z_sk = Z_sk.reshape(XX_sk.shape)

# Plot data
fig, ax = plt.subplots(1, 1, figsize = (10, 10))
ax.imshow(np.rot90(Z_sk), cmap = "RdPu", extent = [min_x_sk, max_x_sk, min_y_sk,
max_y_sk])
ax.plot(x_sk, y_sk, 'k.', markersize = 2, alpha = 0.1)
counties_wgs.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
ax.set_title('San Francisco Bay Area - SciKit-Learn Kernel Density Estimation for
Wells', fontdict = {'fontsize': '15', 'fontweight' : '3'})
plt.show()
```



We can export the raster if necessary.

```
def export_kde_raster(Z, XX, YY, min_x, max_x, min_y, max_y, proj, filename):
    '''Export and save a kernel density raster.'''

    # Flip array vertically and rotate 270 degrees
    Z_export = np.rot90(np.flip(Z, 0), 3)

    # Get resolution
    xres = (max_x - min_x) / len(XX)
    yres = (max_y - min_y) / len(YY)

    # Set transform
    transform = Affine.translation(min_x - xres / 2, min_y - yres / 2) *
Affine.scale(xres, yres)

    # Export array as raster
    with rasterio.open(
            filename,
            mode = "w",
            driver = "GTiff",
            height = Z_export.shape[0],
            width = Z_export.shape[1],
            count = 1,
            dtype = Z_export.dtype,
            crs = proj,
            transform = transform,
    ) as new_dataset:
            new_dataset.write(Z_export, 1)
```

```
# Export raster
export_kde_raster(Z = Z_sk, XX = XX_sk, YY = YY_sk,
                  min_x = min_x_sk, max_x = max_x_sk, min_y = min_y_sk, max_y =
max_y_sk,
                  proj = proj_wgs, filename = "../temp/bay-area-
wells_kde_sklearn.tif")
```

There are a few other ways to compute KDE in Python. This article reviews and compares all these implementations.

---

[1](1,2,3,4) How Kernel Density works, Esri

[2](1,2,3,4) GIS Fundamentals: A First Text on Geographic Information Systems, 5th ed., Paul Bolstad

Spatial Interpolation

Learning Objectives

- Conduct various types of interpolation on point dataset
- Obtain interpolated values at specified unsampled locations

Review

- Spatial Vector Data
- Attributes & Indexing for Vector Data
- Creating Spatial Vector Data
- Merge Data & Dissolve Polygons

Interpolation is the process of using locations with known, sampled values (of a phenomenon) to estimate the values at unknown, unsampled areas [1]. In this chapter, we will explore three interpolation methods: Thiessen polygons (Voronoi diagrams), k-nearest neighbors (KNN), and kriging.

We will first begin by importing modules (click the + below to show code cell).

```
# Import modules
import geopandas as gpd
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from pykrige.ok import OrdinaryKriging
import rasterio
import rasterio.mask
from rasterio.plot import show
from rasterio.transform import Affine
from scipy.spatial import Voronoi, voronoi_plot_2d
from shapely.geometry import box
from shapely.geometry import Polygon, Point
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
```

We will utilize shapefiles of San Francisco Bay Area county boundaries and rainfall “values” that were “sampled” in the Bay Area. We will load in the data and reproject the data (click the + below to show code cell).

Note

It is critical to use a ‘projected’ coordinate system when doing interpolation. If you keep your data in geographic lat lon distances will vary significantly as you move up and down in latitude… since interpolation depends on distance as a way of establishing relationships this would be a problem… a big one.

```
# Load data

# County boundaries
# Source: https://opendata.mtc.ca.gov/datasets/san-francisco-bay-region-counties-clipped?geometry=-125.590%2C37.123%2C-119.152%2C38.640
counties =
gpd.read_file("../_static/e_vector_shapefiles/sf_bay_counties/sf_bay_counties.shp"
)

# Rainfall measurement "locations"
# Source: https://earthworks.stanford.edu/catalog/stanford-td754wr4701
# Modified by author by clipping raster to San Francisco Bay Area, generating random points, and extracting raster values (0-255) to the points
rainfall =
gpd.read_file("../_static/e_vector_shapefiles/sf_bay_rainfall/sf_bay_rainfall.shp"
)

# Reproject data to CA Teale Albert
# https://nrm.dfg.ca.gov/FileHandler.ashx?DocumentID=109326&inline
proj = "+proj=aea +lat_1=34 +lat_2=40.5 +lat_0=0 +lon_0=-120 +x_0=0 +y_0=-4000000 +ellps=GRS80 +datum=NAD83 +units=m +no_defs "
counties = counties.to_crs(proj)
rainfall = rainfall.to_crs(proj)
```

Next, we’ll prepare the data for geoprocessing (click the + below to show code cell).

```
# Get X and Y coordinates of rainfall points
x_rain = rainfall["geometry"].x
y_rain = rainfall["geometry"].y

# Create list of XY coordinate pairs
coords_rain = [list(xy) for xy in zip(x_rain, y_rain)]

# Get extent of counties feature
min_x_counties, min_y_counties, max_x_counties, max_y_counties =
counties.total_bounds

# Get list of rainfall "values"
value_rain = list(rainfall["VALUE"])

# Create a copy of counties dataset
counties_dissolved = counties.copy()

# Add a field with constant value of 1
counties_dissolved["constant"] = 1

# Dissolve all counties to create one polygon
counties_dissolved = counties_dissolved.dissolve(by = "constant").reset_index(drop = True)
```

We will also define a function for exporting rasters.

```
def export_kde_raster(Z, XX, YY, min_x, max_x, min_y, max_y, proj, filename):
    '''Export and save a kernel density raster.'''

    # Get resolution
    xres = (max_x - min_x) / len(XX)
    yres = (max_y - min_y) / len(YY)

    # Set transform
    transform = Affine.translation(min_x - xres / 2, min_y - yres / 2) *
Affine.scale(xres, yres)

    # Export array as raster
    with rasterio.open(
            filename,
            mode = "w",
            driver = "GTiff",
            height = Z.shape[0],
            width = Z.shape[1],
            count = 1,
            dtype = Z.dtype,
            crs = proj,
            transform = transform,
    ) as new_dataset:
            new_dataset.write(Z, 1)
```

With any model used for prediction, it is important to assess the model fit for unobserved locations (or the accuracy of the values predicted by the model in relation to their actual values). Thus, in order to assess the fit, we break our data into two portions, a “training” data set used to train the model, and a “testing” set that remains “unseen” by the model but can be used to assess model performance. Effectively, we can use this “unseen” testing subset to validate the model because we can compare their true values with the estimated value from the model prediction.

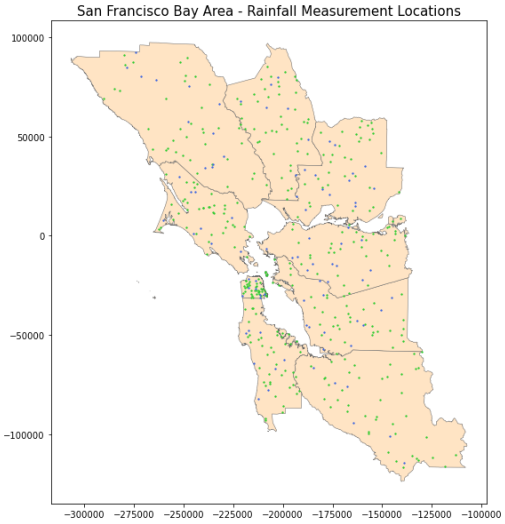
We will separate our rainfall dataset into two subsets: one for training and the other for testing. These subsets will be used in our KNN and kriging analyses.

```
# Split data into testing and training sets
coords_rain_train, coords_rain_test, value_rain_train, value_rain_test =
train_test_split(coords_rain, value_rain, test_size = 0.20, random_state = 42)

# Create separate GeoDataFrames for testing and training sets
rain_train_gdf = gpd.GeoDataFrame(geometry = [Point(x, y) for x, y in
coords_rain_train], crs = proj)
rain_train_gdf["Actual_Value"] = value_rain_train
rain_test_gdf = gpd.GeoDataFrame(geometry = [Point(x, y) for x, y in
coords_rain_test], crs = proj)
rain_test_gdf["Actual_Value"] = value_rain_test

# Get minimum and maximum coordinate values of rainfall training points
min_x_rain, min_y_rain, max_x_rain, max_y_rain = rain_train_gdf.total_bounds
```

Let's plot our data!

```
# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (10, 10))

# Stylize plots
plt.style.use('bmh')

# Plot data
counties.plot(ax = ax, color = 'bisque', edgecolor = 'dimgray')
rain_train_gdf.plot(ax = ax, marker = 'o', color = 'limegreen', markersize = 3)
rain_test_gdf.plot(ax = ax, marker = 'o', color = 'royalblue', markersize = 3)
# Set title
ax.set_title('San Francisco Bay Area - Rainfall Measurement Locations', fontdict =
{'fontsize': '15', 'fontweight' : '3'})
```

```
Text(0.5, 1.0, 'San Francisco Bay Area - Rainfall Measurement Locations')
```



In the map above, the green and blue points are the rainfall points that we loaded separated into the training set and testing set, respectively.

Thiessen Polygons (Voronoi Diagrams)

Thiessen polygons (also known as Voronoi diagrams) polygons allow us to perform nearest neighbor interpolation, which is perhaps the most basic type of interpolation. Thiessen polygons are be constructed around each sampled point so all the space within a specific polygon is closest in distance to that sampled point (as compared to other sampled points). Then, to perform nearest neighbor interpolation, all that space is assigned the value of that sampled point. [1]

We can use the **`scipy`** package to create Thiessen polygons. After running the `Voronoi()` function, we can use the `vertices` attribute to get a list of vertices, which we can subsequently use to generate polygons.

> **Attention**
>
> When creating Thiessen polygons, the sample points toward the edges of the point shapefile's extent will have infinite Voronoi regions, because not all sides of these edge points have adjacent sample points that would constrain the regions. Consequently, these infinite regions will not be exported. To mitigate this issue, we can create dummy points well beyond the extent of our datasets, which will create finite Voronoi regions for all of our actual sample points. Then, we can clip the regions to our extent shapefile (creating dummy points far away from our actual sample points will ensure the dummy points and their infinite Voronoi regions do not interfere with the sample points and their associated finite Voronoi regions after all regions are clipped).

```
# Extend extent of counties feature by using buffer
counties_buffer = counties.buffer(100000)

# Get extent of buffered input feature
min_x_cty_tp, min_y_cty_tp, max_x_cty_tp, max_y_cty_tp =
counties_buffer.total_bounds

# Use extent to create dummy points and add them to list of coordinates
coords_tp = coords_rain_train + [[min_x_cty_tp, min_y_cty_tp], [max_x_cty_tp,
min_y_cty_tp],
                                 [max_x_cty_tp, max_y_cty_tp], [min_x_cty_tp,
max_y_cty_tp]]

# Compute Voronoi diagram
tp = Voronoi(coords_tp)

# Create empty list of hold Voronoi polygons
tp_poly_list = []

# Create a polygon for each region
# 'regions' attribute provides a list of indices of the vertices (in the
'vertices' attribute) that make up the region
# Source:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.Voronoi.html
for region in tp.regions:

    # Ignore region if -1 is in the list (based on documentation)
    if -1 in region:

        # Return to top of loop
        continue

    # Otherwise, pass
    else:
        pass

    # Check that region list has values in it
    if len(region) != 0:

        # Create a polygon by using the region list to call the correct elements
in the 'vertices' attribute
        tp_poly_region = Polygon(list(tp.vertices[region]))

        # Append polygon to list
        tp_poly_list.append(tp_poly_region)

    # If no values, return to top of loop
    else:
        continue

# Create GeoDataFrame from list of polygon regions
tp_polys = gpd.GeoDataFrame(tp_poly_list, columns = ['geometry'], crs = proj)

# Clip polygon regions to the counties boundary
tp_polys_clipped = gpd.clip(tp_polys, counties_dissolved)
```

A spatial join can be conducted to assign the rainfall training “values” to its associated Thiessen polygon.

```
# If rainfall point within the polygon, assign that rainfall value to the polygon
tp_polys_clipped_values = gpd.sjoin(rain_train_gdf, tp_polys_clipped, how =
"right", op = 'within')

# Drop un-needed column
tp_polys_clipped_values = tp_polys_clipped_values.drop("index_left", axis = 1)

# Rename column
tp_polys_clipped_values = tp_polys_clipped_values.rename(columns =
{"Actual_Value": "VALUE_Thiessen"})

# Display head of attribute table
print("Attribute Table: Thiessen Polygon Interpolated Values")
display(tp_polys_clipped_values.head())
```

```
Attribute Table: Thiessen Polygon Interpolated Values
```

	VALUE_Thiessen	geometry
0	31	POLYGON ((-112269.760 -116571.101, -118993.986...
1	30	POLYGON ((-123619.813 -116449.307, -118993.986...
2	58	POLYGON ((-246788.933 78896.506, -260275.835 8...
3	62	POLYGON ((-260275.835 80739.699, -260874.252 8...
4	66	POLYGON ((-237973.581 88639.620, -246524.775 8...

A second spatial join can be conducted to assign those values from the Thiessen polygons to the points from the testing dataset (only if a test point falls within a polygon). We can subsequently get the out-of-sample r-squared value, which is calculated by using the data points that the model did not use (the testing dataset) and comparing the testing dataset’s actual values to the values as predicted by the model.

```
# If test point is within a polygon, assign that polygon's value to the test point
rain_test_pred_tp = gpd.sjoin(rain_test_gdf, tp_polys_clipped_values, how =
"left", op = 'within')

# Drop un-needed column
rain_test_pred_tp = rain_test_pred_tp.drop("index_right", axis = 1)

# Rename column
rain_test_pred_tp = rain_test_pred_tp.rename(columns = {"Actual_Value":
"VALUE_Actual", "VALUE_Thiessen": "VALUE_Predict"})

# Generate out-of-sample R^2
out_r_squared_tp = r2_score(rain_test_pred_tp.VALUE_Actual,
rain_test_pred_tp.VALUE_Predict)
print("Thiessen polygon out-of-sample r-squared:
{}".format(round(out_r_squared_tp, 2)))

# Display attribute table
print("\nAttribute Table: Testing Dataset Interpolated Values - Thiessen Polygon
Method")
display(rain_test_pred_tp.head(2))
```

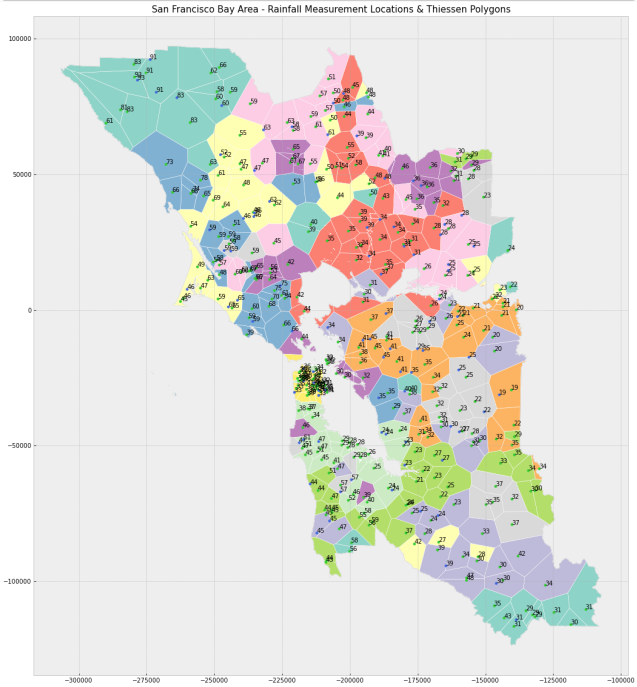
```
Thiessen polygon out-of-sample r-squared: 0.89

Attribute Table: Testing Dataset Interpolated Values - Thiessen Polygon Method
```

	geometry	VALUE_Actual	VALUE_Predict
0	POINT (-229633.026 40063.754)	56	62
1	POINT (-273849.350 92453.689)	73	91

Plotting the data, we see that each polygon has one green training point (and vice-versa). All space within one polygon is closest to the known training point (green dot) within the polygon. The testing points (blue dots) are assigned the value of the Thiessen polygon in which it falls.

```
# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (20, 20))

# Stylize plots
plt.style.use('bmh')

# Plot data
counties_dissolved.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
tp_polys_clipped.plot(ax = ax, cmap = 'Set3', edgecolor = 'white', linewidth =
0.5)
rain_train_gdf.plot(ax = ax, marker = 'o', color = 'limegreen', markersize = 15)
rain_test_pred_tp.plot(ax = ax, marker = 'o', color = 'royalblue', markersize =
15)

# Iterate through each rainfall train point to add a label with its value to the
plot
for index, row in rain_train_gdf.iterrows():
    plt.annotate(row.Actual_Value, (row.geometry.x, row.geometry.y))

# Iterate through each rainfall test point to add a label with its value to the
plot
for index, row in rain_test_pred_tp.iterrows():
    plt.annotate(row.VALUE_Predict, (row.geometry.x, row.geometry.y))

# Set title
ax.set_title('San Francisco Bay Area - Rainfall Measurement Locations & Thiessen
Polygons', fontdict = {'fontsize': '15', 'fontweight' : '3'})
```

```
Text(0.5, 1.0, 'San Francisco Bay Area - Rainfall Measurement Locations & Thiessen
Polygons')
```



In addition to `vertices`, there are a few other attributes we can call if we want to further explore the polygons. These attributes will provide actual values (e.g., vertices) or provide the indices for querying other attributes. [2]

In the example below, we demonstrate how to extract the value of one of the Thiessen polygons at a new location for which we want a predicted value. We use the `point_region` attribute to provide the index of a point's Voronoi region, and we use that index to get the region in `regions`. That provides indices of the vertices that make up the polygon, which we use to get the appropriate values in `vertices`.

```
# Set index for feature of interest
feature_index_one = 5

# Get a Voronoi polygon for one feature
# 'point_region' attribute provides the index of the Voronoi region belonging to a
specified point
# Can use the index to call the appropriate element in the 'regions' attribute
tp_poly_region_one =
Polygon(tp.vertices[tp.regions[tp.point_region[feature_index_one]]])

# Create GeoDataFrame for polygon
tp_poly_region_one = gpd.GeoDataFrame([tp_poly_region_one], columns =
['geometry'], crs = proj)

# Clip polygon to county boundary
tp_poly_region_one = gpd.clip(tp_poly_region_one, counties_dissolved)

# Get the equivalent feature from the rainfall dataset
rain_one = rain_train_gdf.iloc[[feature_index_one]]

# Add the rainfall value to the polygon attribute table
tp_poly_region_one["VALUE_Predict"] = rain_one["Actual_Value"].values

# Display attribute table
print("Attribute Table: Thiessen Polygon Interpolated Value")
display(tp_poly_region_one)
```

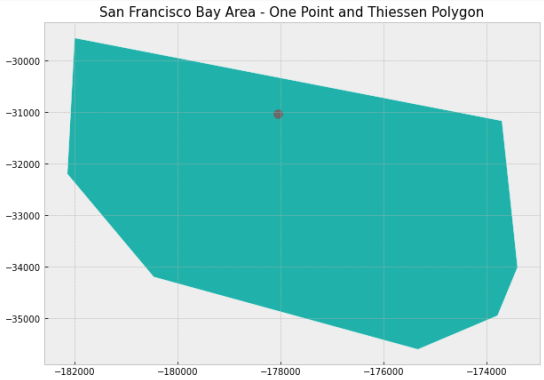
```
Attribute Table: Thiessen Polygon Interpolated Value
```

	geometry	VALUE_Predict
0	POLYGON ((-173793.383 -34945.420, -175342.667 ...	38

Here's how that one Thiessen polygon looks.

```
# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (10, 10))

# Stylize plots
plt.style.use('bmh')

# Plot data
tp_poly_region_one.plot(ax = ax, color = 'lightseagreen', edgecolor = 'white',
linewidth = 0.5)
rain_one.plot(ax = ax, marker = 'o', color = 'dimgray', markersize = 100)

# Set title
ax.set_title('San Francisco Bay Area - One Point and Thiessen Polygon', fontdict =
{'fontsize': '15', 'fontweight' : '3'})
```

```
Text(0.5, 1.0, 'San Francisco Bay Area - One Point and Thiessen Polygon')
```



K-Nearest Neighbors

KNN (also stylized as kNN) is a neighbor-based learning method that can be used for interpolation. Unlike the Thiessen polygons method, KNN looks for a specified number `k` of sampled points closest to an unknown point. The `k` known points can be used to predict the value (discrete or continuous) of the unknown point. [3]

We can use the **`scikit-learn`** module to perform KNN analysis.

```
# Set number of neighbors to look for
neighbors = 5

# Initialize KNN regressor
knn_regressor = KNeighborsRegressor(n_neighbors = neighbors, weights = "distance")

# Fit regressor to data
knn_regressor.fit(coords_rain_train, value_rain_train)
```

```
KNeighborsRegressor(weights='distance')
```

Now that we have created the KNN model, we can get the in-sample r-squared value. An in-sample statistic, as suggested by its name, is calculated by using the data that were used to build the model (the “training” dataset).

```
# Generate in-sample R^2
in_r_squared_knn = knn_regressor.score(coords_rain_train, value_rain_train)
print("KNN in-sample r-squared: {}".format(round(in_r_squared_knn, 2)))
```

```
KNN in-sample r-squared: 1.0
```

Here, the in-sample r-squared value is 100% because KNN is a “exact interpolator.” For exact interpolators, estimated values for known points are exactly equal to actual values. Other methods like Kriging, shown below, are inexact interpolators. For inexact interpolators, estimated values for known points are not exactly equal to actual values. Here’s a visual of inexact versus exact interpolators.

Similarily, we can also get the out-of-sample r-squared value and compare the test dataset’s actual values to the values as predicted by the model.

```
# Generate out-of-sample R^2
out_r_squared_knn = knn_regressor.score(coords_rain_test, value_rain_test)
print("KNN out-of-sample r-squared: {}".format(round(out_r_squared_knn, 2)))

# Predict values for testing dataset
coords_rain_test_predict_knn = knn_regressor.predict(coords_rain_test)

# Create dictionary holding the actual and predicted values
predict_dict_knn = {"Coordinate_Pair": coords_rain_test, "VALUE_Actual":
value_rain_test, "VALUE_Predict": coords_rain_test_predict_knn}

# Create dataframe from dictionary
predict_df_knn = pd.DataFrame(predict_dict_knn)

# Display attribute table
print("\nAttribute Table: Testing Set Interpolated Values - KNN Method")
display(predict_df_knn.head(2))
```

```
KNN out-of-sample r-squared: 0.93

Attribute Table: Testing Set Interpolated Values - KNN Method
```

	Coordinate_Pair	VALUE_Actual	VALUE_Predict
0	[-229633.02624581114, 40063.75429398427]	56	55.827992
1	[-273849.3497429455, 92453.68931061402]	73	87.640078

Out-of-sample r-squared looks pretty strong!

> **Tip**
>
> If you are just interested in identifying the `k` nearest neighbors (no interpolation), use the **`NearestNeighbors()`** function.

Kriging

Kriging is a type of interpolation that uses a semivariogram, which measures spatial autocorrelation (how similar close points are in value and how this similarity changes as distance between points increases). Thus, the semivariogram determines how much influence a known point has on an unknown point as the distance between the known point and the unknown point increases. In other words, the weight of a known point on an unknown point decreases with increasing distance, and the semivariogram determines how quickly that weight tapers with increasing distance. [1], [4]

For more information, see this ArcGIS help guide on kriging.

Two Python packages that can be used for kriging include `scikit-learn` and `pykrige`. The former package works best when the input data has a WGS 84 projection, so we will begin by reprojecting all of our data to that coordinate system (click the + below to show code cell).

```
'''
# Set projection to WGS 84 and reproject data
proj_wgs = 4326
counties_wgs = counties.to_crs(proj_wgs)
rainfall_wgs = rainfall.to_crs(proj_wgs)
rain_train_gdf_wgs = rain_train_gdf.to_crs(proj_wgs)
rain_test_gdf_wgs = rain_test_gdf.to_crs(proj_wgs)

# Get X and Y coordinates of rainfall points
x_rain_wgs = rainfall_wgs["geometry"].x
y_rain_wgs = rainfall_wgs["geometry"].y

# Create list of XY coordinate pairs
coords_rain_train_wgs = [list(xy) for xy in zip(rain_train_gdf_wgs["geometry"].x,
rain_train_gdf_wgs["geometry"].y)]
coords_rain_test_wgs = [list(xy) for xy in zip(rain_test_gdf_wgs["geometry"].x,
rain_test_gdf_wgs["geometry"].y)]

# Get minimum and maximum coordinate values of rainfall points
min_x_rain_wgs, min_y_rain_wgs, max_x_rain_wgs, max_y_rain_wgs =
rain_train_gdf_wgs.total_bounds
'''
```

```
'\n# Set projection to WGS 84 and reproject data\nproj_wgs = 4326\ncounties_wgs =
counties.to_crs(proj_wgs)\nrainfall_wgs =
rainfall.to_crs(proj_wgs)\nrain_train_gdf_wgs =
rain_train_gdf.to_crs(proj_wgs)\nrain_test_gdf_wgs =
rain_test_gdf.to_crs(proj_wgs)\n\n# Get X and Y coordinates of rainfall
points\nx_rain_wgs = rainfall_wgs["geometry"].x\ny_rain_wgs =
rainfall_wgs["geometry"].y\n\n# Create list of XY coordinate
pairs\ncoords_rain_train_wgs = [list(xy) for xy in
zip(rain_train_gdf_wgs["geometry"].x,
rain_train_gdf_wgs["geometry"].y)]\ncoords_rain_test_wgs = [list(xy) for xy in
zip(rain_test_gdf_wgs["geometry"].x, rain_test_gdf_wgs["geometry"].y)]\n\n# Get
minimum and maximum coordinate values of rainfall points\nmin_x_rain_wgs,
min_y_rain_wgs, max_x_rain_wgs, max_y_rain_wgs =
rain_train_gdf_wgs.total_bounds\n'
```

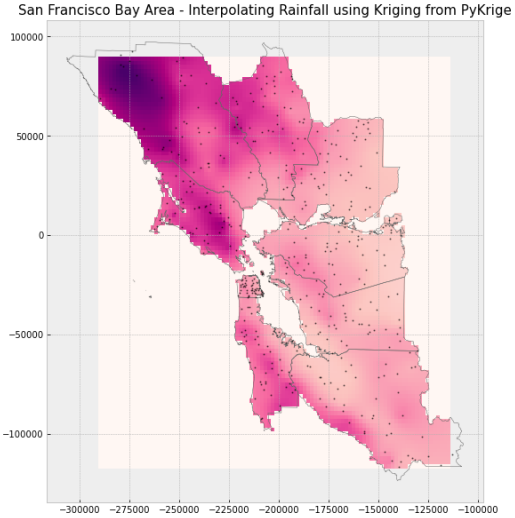
Method 1 - Using `PyKrige`

The **`pykrige`** module offers ordinary and universal kriging. It also supports various variogram models in addition to Gaussian.

```
# Adapted from: https://geostat-framework.readthedocs.io/projects/pykrige/en/latest/examples/04_krige_geometric.html

# Create a 100 by 100 grid
# Horizontal and vertical cell counts should be the same
XX_pk_krig = np.linspace(min_x_rain, max_x_rain, 100)
YY_pk_krig = np.linspace(min_y_rain, max_y_rain, 100)

# Generate ordinary kriging object
OK = OrdinaryKriging(
    np.array(x_rain),
    np.array(y_rain),
    value_rain,
    variogram_model = "linear",
    verbose = False,
    enable_plotting = False,
    coordinates_type = "euclidean",
)

# Evaluate the method on grid
Z_pk_krig, sigma_squared_p_krig = OK.execute("grid", XX_pk_krig, YY_pk_krig)

# Export raster
export_kde_raster(Z = Z_pk_krig, XX = XX_pk_krig, YY = YY_pk_krig,
                  min_x = min_x_rain, max_x = max_x_rain, min_y = min_y_rain,
max_y = max_y_rain,
                  proj = proj, filename = "../temp/e_bay-area-
rain_pk_kriging.tif")

# Open raster
raster_pk = rasterio.open("../temp/e_bay-area-rain_pk_kriging.tif")


# Create polygon with extent of raster
poly_shapely = box(*raster_pk.bounds)

# Create a dictionary with needed attributes and required geometry column
attributes_df = {'Attribute': ['name1'], 'geometry': poly_shapely}

# Convert shapely object to a GeoDataFrame
raster_pk_extent = gpd.GeoDataFrame(attributes_df, geometry = 'geometry', crs =
proj)

# Create copy of test dataset
rain_test_gdf_pk_krig = rain_test_gdf.copy()

# Subset the GeoDataFrame by checking which test points are within the raster
extent polygon
# If a test point is beyond the extent of training points dataset, the kriging
output may not cover that test point
rain_test_gdf_pk_krig =
rain_test_gdf_pk_krig[rain_test_gdf_pk_krig.within(raster_pk_extent.geometry.value
s[0])]

# Create list of XY coordinate pairs for the test points that fall within raster
extent polygon
coords_rain_test_pk_krig = [list(xy) for xy in
zip(rain_test_gdf_pk_krig["geometry"].x, rain_test_gdf_pk_krig["geometry"].y)]

# Extract raster value at each test point and add the values to the GeoDataFrame
rain_test_gdf_pk_krig["VALUE_Predict"] = [x[0] for x in
raster_pk.sample(coords_rain_test_pk_krig)]

# Generate out-of-sample R^2
out_r_squared_tp = r2_score(rain_test_gdf_pk_krig.Actual_Value,
rain_test_gdf_pk_krig.VALUE_Predict)
print("PyKrige Kriging out-of-sample r-squared: {}".format(round(out_r_squared_tp,
2)))

# Display attribute table
print("\nAttribute Table: Random Points Interpolated Values - PyKrige Kriging
Method")
display(rain_test_gdf_pk_krig.head(2))


# Mask raster to counties shape
out_image_pk, out_transform_pk = rasterio.mask.mask(raster_pk,
counties.geometry.values, crop = True)

# Stylize plots
plt.style.use('bmh')

# Plot data
fig, ax = plt.subplots(1, figsize = (10, 10))
show(out_image_pk, ax = ax, transform = out_transform_pk, cmap = "RdPu")
ax.plot(x_rain, y_rain, 'k.', markersize = 2, alpha = 0.5)
counties.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
plt.gca().invert_yaxis()

# Set title
ax.set_title('San Francisco Bay Area - Interpolating Rainfall using Kriging from
PyKrige', fontdict = {'fontsize': '15', 'fontweight' : '3'})

# Display plot
plt.show()
```

```
PyKrige Kriging out-of-sample r-squared: 0.99

Attribute Table: Random Points Interpolated Values - PyKrige Kriging Method
```

	geometry	Actual_Value	VALUE_Predict
0	POINT (-229633.026 40063.754)	56	55.089479
2	POINT (-244809.950 1296.393)	62	61.013348



Method 2- Using `scikit-learn`

Kriging can be performed using Gaussian processes from the **`scikit-learn`** module (Gaussian processes is essentially equivalent to kriging). Various kernels for Gaussian processes can be specified. We will continue to use the training and testing datasets created from our KNN analysis.

```
'''
# Create a 100 by 100 cell mesh grid
# Horizontal and vertical cell counts should be the same
XX_sk_krig, YY_sk_krig = np.mgrid[min_x_rain_wgs:max_x_rain_wgs:100j,
min_y_rain_wgs:max_y_rain_wgs:100j]

# Create 2-D array of the coordinates (paired) of each cell in the mesh grid
positions_sk_krig = np.vstack([XX_sk_krig.ravel(), YY_sk_krig.ravel()]).T

# Generate Gaussian Process model (can change parameters as desired)
gp = GaussianProcessRegressor(n_restarts_optimizer = 10)

# Fit kernel density estimator to coordinates and values
gp.fit(coords_rain_train_wgs, value_rain_train)

# Evaluate the model on coordinate pairs
Z_sk_krig = gp.predict(positions_sk_krig)

# Reshape the data to fit mesh grid
Z_sk_krig = Z_sk_krig.reshape(XX_sk_krig.shape)
'''
```

```
'\n# Create a 100 by 100 cell mesh grid\n# Horizontal and vertical cell counts
should be the same\nXX_sk_krig, YY_sk_krig =
np.mgrid[min_x_rain_wgs:max_x_rain_wgs:100j,
min_y_rain_wgs:max_y_rain_wgs:100j]\n\n# Create 2-D array of the coordinates
(paired) of each cell in the mesh grid\npositions_sk_krig =
np.vstack([XX_sk_krig.ravel(), YY_sk_krig.ravel()]).T\n\n# Generate Gaussian
Process model (can change parameters as desired)\ngp =
GaussianProcessRegressor(n_restarts_optimizer = 10)\n\n# Fit kernel density
estimator to coordinates and values\ngp.fit(coords_rain_train_wgs,
value_rain_train)\n\n# Evaluate the model on coordinate pairs\nZ_sk_krig =
gp.predict(positions_sk_krig)\n\n# Reshape the data to fit mesh grid\nZ_sk_krig =
Z_sk_krig.reshape(XX_sk_krig.shape)\n'
```

Next, we can calculate our r-squared statistics and predictions.

```
'''
# Generate in-sample R^2
in_r_squared_sk_krig = gp.score(coords_rain_train_wgs, value_rain_train)
print("Scikit-Learn Kriging in-sample r-squared:
{}".format(round(in_r_squared_sk_krig, 2)))

# Generate out-of-sample R^2
out_r_squared_sk_krig = gp.score(coords_rain_test_wgs, value_rain_test)
print("Scikit-Learn Kriging out-of-sample r-squared:
{}".format(round(out_r_squared_sk_krig, 2)))

# Predict values for testing dataset
coords_rain_test_predict_sk_krig = gp.predict(coords_rain_test_wgs)

# Create dictionary holding the actual and predicted values
predict_dict_sk_krig = {"Coordinate_Pair": coords_rain_test_wgs, "VALUE_Actual":
value_rain_test, "VALUE_Predict": coords_rain_test_predict_sk_krig}

# Create dataframe from dictionary
predict_df_sk_krig = pd.DataFrame(predict_dict_sk_krig)

# Display attribute table
print("\nAttribute Table: Testing Set Interpolated Values - Scikit-Learn Kriging
Method")
display(predict_df_sk_krig.head(2))
'''
```

```
'\n# Generate in-sample R^2\nin_r_squared_sk_krig =
gp.score(coords_rain_train_wgs, value_rain_train)\nprint("Scikit-Learn Kriging in-
sample r-squared: {}".format(round(in_r_squared_sk_krig, 2)))\n\n# Generate out-
of-sample R^2\nout_r_squared_sk_krig = gp.score(coords_rain_test_wgs,
value_rain_test)\nprint("Scikit-Learn Kriging out-of-sample r-squared:
{}".format(round(out_r_squared_sk_krig, 2)))\n\n# Predict values for testing
dataset\ncoords_rain_test_predict_sk_krig = gp.predict(coords_rain_test_wgs)\n\n#
Create dictionary holding the actual and predicted values\npredict_dict_sk_krig =
{"Coordinate_Pair": coords_rain_test_wgs, "VALUE_Actual": value_rain_test,
"VALUE_Predict": coords_rain_test_predict_sk_krig}\n\n# Create dataframe from
dictionary\npredict_df_sk_krig = pd.DataFrame(predict_dict_sk_krig)\n\n# Display
attribute table\nprint("\nAttribute Table: Testing Set Interpolated Values -
Scikit-Learn Kriging Method")\ndisplay(predict_df_sk_krig.head(2))\n'
```

Model seems like a good fit! Let's export the raster.

```
'''
# Flip array vertically and rotate 270 degrees
Z_sk_krig = np.rot90(np.flip(Z_sk_krig, 0), 3)

# Export raster
export_kde_raster(Z = Z_sk_krig, XX = XX_sk_krig, YY = YY_sk_krig,
                  min_x = min_x_rain_wgs, max_x = max_x_rain_wgs, min_y =
min_y_rain_wgs, max_y = max_y_rain_wgs,
                  proj = proj_wgs, filename = ../temp/e_bay-area-
rain_sk_kriging.tif")
'''
```

```
'\n# Flip array vertically and rotate 270 degrees\nZ_sk_krig =
np.rot90(np.flip(Z_sk_krig, 0), 3)\n\n# Export raster\nexport_kde_raster(Z =
Z_sk_krig, XX = XX_sk_krig, YY = YY_sk_krig,\n                  min_x =
min_x_rain_wgs, max_x = max_x_rain_wgs, min_y = min_y_rain_wgs, max_y =
max_y_rain_wgs,\n                  proj = proj_wgs, filename = ../temp/e_bay-area-
rain_sk_kriging.tif")\n'
```

Attention

The resulting raster should be clipped. Because the resulting raster covers the extent of the points in a bounding box fashion, the raster in this case covers areas that are not within the counties boundaries (such as in the ocean) where we do not have sample points. Thus, there will be interpolated values in those areas that might not make sense.

Finally, we import the raster, mask it to the counties boundaries, and plot the data.

```
'''
# Open raster
raster_sk = rasterio.open("../temp/e_bay-area-rain_sk_kriging.tif")

# Mask raster to counties shape
out_image_sk, out_transform_sk = rasterio.mask.mask(raster_sk,
counties_wgs.geometry.values, crop = True)

# Stylize plots
plt.style.use('bmh')
'''
```

```
'\n# Open raster\nraster_sk = rasterio.open("../temp/e_bay-area-
rain_sk_kriging.tif")\n\n# Mask raster to counties shape\nout_image_sk,
out_transform_sk = rasterio.mask.mask(raster_sk, counties_wgs.geometry.values,
crop = True)\n\n# Stylize plots\nplt.style.use(\'bmh\')\n'
```

Method 2- Using `scikit-learn`

Kriging can be performed using Gaussian processes from the **`scikit-learn`** module (Gaussian processes is essentially equivalent to kriging). Various kernels for Gaussian processes can be specified. We will continue to use the training and testing datasets created from our KNN analysis.

```
'''
# Create a 100 by 100 cell mesh grid
# Horizontal and vertical cell counts should be the same
XX_sk_krig, YY_sk_krig = np.mgrid[min_x_rain_wgs:max_x_rain_wgs:100j,
min_y_rain_wgs:max_y_rain_wgs:100j]

# Create 2-D array of the coordinates (paired) of each cell in the mesh grid
positions_sk_krig = np.vstack([XX_sk_krig.ravel(), YY_sk_krig.ravel()]).T

# Generate Gaussian Process model (can change parameters as desired)
gp = GaussianProcessRegressor(n_restarts_optimizer = 10)

# Fit kernel density estimator to coordinates and values
gp.fit(coords_rain_train_wgs, value_rain_train)

# Evaluate the model on coordinate pairs
Z_sk_krig = gp.predict(positions_sk_krig)

# Reshape the data to fit mesh grid
Z_sk_krig = Z_sk_krig.reshape(XX_sk_krig.shape)
'''
```

```
'\n# Create a 100 by 100 cell mesh grid\n# Horizontal and vertical cell counts
should be the same\nXX_sk_krig, YY_sk_krig =
np.mgrid[min_x_rain_wgs:max_x_rain_wgs:100j,
min_y_rain_wgs:max_y_rain_wgs:100j]\n\n# Create 2-D array of the coordinates
(paired) of each cell in the mesh grid\npositions_sk_krig =
np.vstack([XX_sk_krig.ravel(), YY_sk_krig.ravel()]).T\n\n# Generate Gaussian
Process model (can change parameters as desired)\ngp =
GaussianProcessRegressor(n_restarts_optimizer = 10)\n\n# Fit kernel density
estimator to coordinates and values\ngp.fit(coords_rain_train_wgs,
value_rain_train)\n\n# Evaluate the model on coordinate pairs\nZ_sk_krig =
gp.predict(positions_sk_krig)\n\n# Reshape the data to fit mesh grid\nZ_sk_krig =
Z_sk_krig.reshape(XX_sk_krig.shape)\n'
```

Next, we can calculate our r-squared statistics and predictions.

```
'''
# Generate in-sample R^2
in_r_squared_sk_krig = gp.score(coords_rain_train_wgs, value_rain_train)
print("Scikit-Learn Kriging in-sample r-squared:
{}".format(round(in_r_squared_sk_krig, 2)))

# Generate out-of-sample R^2
out_r_squared_sk_krig = gp.score(coords_rain_test_wgs, value_rain_test)
print("Scikit-Learn Kriging out-of-sample r-squared:
{}".format(round(out_r_squared_sk_krig, 2)))

# Predict values for testing dataset
coords_rain_test_predict_sk_krig = gp.predict(coords_rain_test_wgs)

# Create dictionary holding the actual and predicted values
predict_dict_sk_krig = {"Coordinate_Pair": coords_rain_test_wgs, "VALUE_Actual":
value_rain_test, "VALUE_Predict": coords_rain_test_predict_sk_krig}

# Create dataframe from dictionary
predict_df_sk_krig = pd.DataFrame(predict_dict_sk_krig)

# Display attribute table
print("\nAttribute Table: Testing Set Interpolated Values - Scikit-Learn Kriging
Method")
display(predict_df_sk_krig.head(2))
'''
```

```
'\n# Generate in-sample R^2\nin_r_squared_sk_krig =
gp.score(coords_rain_train_wgs, value_rain_train)\nprint("Scikit-Learn Kriging in-
sample r-squared: {}".format(round(in_r_squared_sk_krig, 2)))\n\n# Generate out-
of-sample R^2\nout_r_squared_sk_krig = gp.score(coords_rain_test_wgs,
value_rain_test)\nprint("Scikit-Learn Kriging out-of-sample r-squared:
{}".format(round(out_r_squared_sk_krig, 2)))\n\n# Predict values for testing
dataset\ncoords_rain_test_predict_sk_krig = gp.predict(coords_rain_test_wgs)\n\n#
Create dictionary holding the actual and predicted values\npredict_dict_sk_krig =
{"Coordinate_Pair": coords_rain_test_wgs, "VALUE_Actual": value_rain_test,
"VALUE_Predict": coords_rain_test_predict_sk_krig}\n\n# Create dataframe from
dictionary\npredict_df_sk_krig = pd.DataFrame(predict_dict_sk_krig)\n\n# Display
attribute table\nprint("\nAttribute Table: Testing Set Interpolated Values -
Scikit-Learn Kriging Method")\ndisplay(predict_df_sk_krig.head(2))\n'
```

Model seems like a good fit! Let's export the raster.

```
'''
# Flip array vertically and rotate 270 degrees
Z_sk_krig = np.rot90(np.flip(Z_sk_krig, 0), 3)

# Export raster
export_kde_raster(Z = Z_sk_krig, XX = XX_sk_krig, YY = YY_sk_krig,
                  min_x = min_x_rain_wgs, max_x = max_x_rain_wgs, min_y =
min_y_rain_wgs, max_y = max_y_rain_wgs,
                  proj = proj_wgs, filename = ../temp/e_bay-area-
rain_sk_kriging.tif")
'''
```

```
'\n# Flip array vertically and rotate 270 degrees\nZ_sk_krig =
np.rot90(np.flip(Z_sk_krig, 0), 3)\n\n# Export raster\nexport_kde_raster(Z =
Z_sk_krig, XX = XX_sk_krig, YY = YY_sk_krig,\n                  min_x =
min_x_rain_wgs, max_x = max_x_rain_wgs, min_y = min_y_rain_wgs, max_y =
max_y_rain_wgs,\n                  proj = proj_wgs, filename = ../temp/e_bay-area-
rain_sk_kriging.tif")\n'
```

Attention

The resulting raster should be clipped. Because the resulting raster covers the extent of the points in a bounding box fashion, the raster in this case covers areas that are not within the counties boundaries (such as in the ocean) where we do not have sample points. Thus, there will be interpolated values in those areas that might not make sense.

Finally, we import the raster, mask it to the counties boundaries, and plot the data.

```
'''
# Open raster
raster_sk = rasterio.open("../temp/e_bay-area-rain_sk_kriging.tif")

# Mask raster to counties shape
out_image_sk, out_transform_sk = rasterio.mask.mask(raster_sk,
counties_wgs.geometry.values, crop = True)

# Stylize plots
plt.style.use('bmh')

# Plot data
fig, ax = plt.subplots(1, figsize = (10, 10))
show(out_image_sk, ax = ax, transform = out_transform_sk, cmap = "RdPu")
ax.plot(x_rain_wgs, y_rain_wgs, 'k.', markersize = 2, alpha = 0.5)
counties_wgs.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
plt.gca().invert_yaxis()

# Set title
ax.set_title('San Francisco Bay Area - Interpolating Rainfall using Kriging from
Scikit-Learn', fontdict = {'fontsize': '15', 'fontweight' : '3'})

# Display plot
plt.show()


# Plot data
fig, ax = plt.subplots(1, figsize = (10, 10))
show(out_image_sk, ax = ax, transform = out_transform_sk, cmap = "RdPu")
ax.plot(x_rain_wgs, y_rain_wgs, 'k.', markersize = 2, alpha = 0.5)
counties_wgs.plot(ax = ax, color = 'none', edgecolor = 'dimgray')
plt.gca().invert_yaxis()

# Set title
ax.set_title('San Francisco Bay Area - Interpolating Rainfall using Kriging from
Scikit-Learn', fontdict = {'fontsize': '15', 'fontweight' : '3'})

# Display plot
plt.show()
'''
```

```
'\n# Open raster\nraster_sk = rasterio.open("../temp/e_bay-area-
rain_sk_kriging.tif")\n\n# Mask raster to counties shape\nout_image_sk,
out_transform_sk = rasterio.mask.mask(raster_sk, counties_wgs.geometry.values,
crop = True)\n\n# Stylize plots\nplt.style.use(\'bmh\')\n\n# Plot data\nfig, ax =
plt.subplots(1, figsize = (10, 10))\nshow(out_image_sk, ax = ax, transform =
out_transform_sk, cmap = "RdPu")\nax.plot(x_rain_wgs, y_rain_wgs, \'k.\',
markersize = 2, alpha = 0.5)\ncounties_wgs.plot(ax = ax, color = \'none\',
edgecolor = \'dimgray\')\nplt.gca().invert_yaxis()\n\n# Set
title\nax.set_title(\'San Francisco Bay Area - Interpolating Rainfall using
Kriging from Scikit-Learn\', fontdict = {\'fontsize\': \'15\', \'fontweight\' :
\'3\'})\n\n# Display plot\nplt.show()\n\n\n# Plot data\nfig, ax = plt.subplots(1,
figsize = (10, 10))\nshow(out_image_sk, ax = ax, transform = out_transform_sk,
cmap = "RdPu")\nax.plot(x_rain_wgs, y_rain_wgs, \'k.\', markersize = 2, alpha =
0.5)\ncounties_wgs.plot(ax = ax, color = \'none\', edgecolor =
\'dimgray\')\nplt.gca().invert_yaxis()\n\n# Set title\nax.set_title(\'San
Francisco Bay Area - Interpolating Rainfall using Kriging from Scikit-Learn\',
fontdict = {\'fontsize\': \'15\', \'fontweight\' : \'3\'})\n\n# Display
plot\nplt.show()\n'
```


[1](1,2,3) GIS Fundamentals: A First Text on Geographic Information Systems, 5th ed., Paul Bolstad

[2] scipy.spatial.Voronoi, SciPy

[3] Nearest Neighbors, scikit-learn

[4] How Kriging works, Esri

Learning Objectives

- Creating raster data with Rasterio
- Creating raster data with Rasterio

Review

- Affine transformation
- Raster Coordinate Reference Systems
- Spatial Raster Data

Reading & Writing Rasters with Rasterio

In order to work with raster data we will be using `rasterio` and later `geowombat`. Behind the scenes a `numpy.ndarray` does all the heavy lifting. To understand how raster works it helps to construct one from scratch.

Here we create two `ndarray` objects one `X` spans [-90°,90°] longitude, and `Y` covers [-90°,90°] latitude.

```
import numpy as np
x = np.linspace(-90, 90, 6)
y = np.linspace(90, -90, 6)
X, Y = np.meshgrid(x, y)
X
```

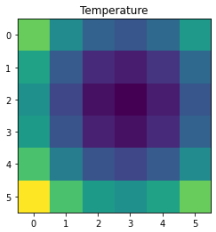
```
array([[-90., -54., -18.,  18.,  54.,  90.],
       [-90., -54., -18.,  18.,  54.,  90.],
       [-90., -54., -18.,  18.,  54.,  90.],
       [-90., -54., -18.,  18.,  54.,  90.],
       [-90., -54., -18.,  18.,  54.,  90.],
       [-90., -54., -18.,  18.,  54.,  90.]])
```

```
Y
```

```
array([[ 90.,  90.,  90.,  90.,  90.,  90.],
       [ 54.,  54.,  54.,  54.,  54.,  54.],
       [ 18.,  18.,  18.,  18.,  18.,  18.],
       [-18., -18., -18., -18., -18., -18.],
       [-54., -54., -54., -54., -54., -54.],
       [-90., -90., -90., -90., -90., -90.]])
```

Let's generate some data representing temperature and store it an array `Z`

```
import matplotlib.pyplot as plt

Z1 =  np.abs(((X - 10) ** 2 + (Y - 10) ** 2) / 1 ** 2)
Z2 =  np.abs(((X + 10) ** 2 + (Y + 10) ** 2) / 2.5 ** 2)
Z =  (Z1 - Z2)

plt.imshow(Z)
plt.title("Temperature")
plt.show()
```



Creating Raster Data

The final array `Z` still lacks a number of elements that transform it from being a non-spatial `numpy` array to a spatial one usable by `rasterio` etc. `Rasterio` requires the following elements to write out a geotif, or spatial raster dataset:

Parameters	Description	Argument
driver	the name of the desired format driver	`'GTiff'`
width	the number of columns of the dataset	`Z.shape[1]`
height	the number of rows of the dataset	`Z.shape[0]`
count	a count of the dataset bands	`1`
dtype	the data type of the dataset	`Z.dtype`
crs	a coordinate reference system identifier or description	`'+proj=latlong'`
transform	an affine transformation matrix	`Affine.translation(x[0] - xres / 2, y[0] - yres / 2) * Affine.scale(xres, yres)`
nodata	a “nodata” value	`-9999`

> **Note**
>
> `transform` defines the location of the upper left hand corner of the raster on the globe, and its spatial resolution. The arguments for `transform` are complex and beyond the scope of the chapter, please refer to the next chapter @ affine transforms and raster crs for more info.

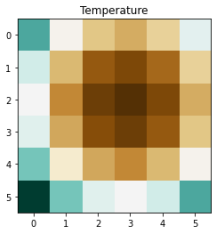
Writing Rasters

To save this array along with spatial information to a file, we call `rasterio.open()` with a path to the new file to be created, and ‘w’ to specify writing mode, along with the arguments above.

```
import rasterio
from rasterio.transform import Affine

xres = (x[-1] - x[0]) / len(x)
yres = (y[-1] - y[0]) / len(y)

transform = Affine.translation(x[0] - xres / 2, y[0] - yres / 2) *
Affine.scale(xres, yres)

with rasterio.open(
        "../temp/temperature.tif",
        mode="w",
        driver="GTiff",
        height=Z.shape[0],
        width=Z.shape[1],
        count=1,
        dtype=Z.dtype,
        crs="+proj=latlong",
        transform=transform,
) as new_dataset:
        new_dataset.write(Z, 1)
```

When we go to look again at our data we can read it back in using `open`. Before plotting we `raster.read(1)` the first band, which converts the data back to a `numpy` array that can be plotted in `matplotlib`. Notice that we also specify the band number (in this case 1), with `.write(Z, 1)`.

```
import matplotlib.pyplot as plt

raster = rasterio.open("../temp/temperature.tif")

plt.imshow(raster.read(1), cmap="BrBG")
plt.title("Temperature")
plt.show()
```



So in summary, a spatial dataset is essentially just a `numpy.ndarray` with information about the location and resolution of the array, the coordinate reference system, and number of bands. This information is typically accessed and updated via the `.profile`.

```
print(raster.profile)
```

```
{'driver': 'GTiff', 'dtype': 'float64', 'nodata': None, 'width': 6, 'height': 6,
'count': 1, 'crs': CRS.from_epsg(4326), 'transform': Affine(30.0, 0.0, -105.0,
       0.0, -30.0, 105.0), 'tiled': False, 'interleave': 'band'}
```

Update Raster Metadata

Notice the the `.profile` above is missing a meaningful `nodata` value and was uncompressed. Let’s learn how to update these values. For this we can update the `.profile` dictionary.

```
# start with the original profile
profile = raster.profile
# update values for nodata and compression type
profile.update(nodata=0, compress="lzw")
print(profile)
```

```
{'driver': 'GTiff', 'dtype': 'float64', 'nodata': 0, 'width': 6, 'height': 6,
'count': 1, 'crs': CRS.from_epsg(4326), 'transform': Affine(30.0, 0.0, -105.0,
       0.0, -30.0, 105.0), 'tiled': False, 'interleave': 'band', 'compress':
'lzw'}
```

Now we just need to write the array of data (obtained from `raster.read(1)`) out with the updated profile info. We can unpack all the dictionary values from the profile using `**profile`.

```
with rasterio.open("../temp/temperature.tif",
                   mode="w",
                   **profile,) as update_dataset:
    update_dataset.write(raster.read(1), 1)
```

Rasterio Multiband Rasters

Working with multiband imagery starts to get a bit tricky, especially with `rasterio` alone.

Let's start with a problematic raster file, a landsat image that stores its red, green, and blue bands in reverse order (blue, green, red), that is scaled by 100 (multiplied by 100 to store data as integers rather than float), and that is missing a meaningful nodata value.

```
with rasterio.open(
    "../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF", mode="r", nodata=0) as
src:
    # read in the array, band 3 first, then band 2, then band 1
    arr = src.read([3, 2, 1])
    # the array has three bands
    print("Array shape:", arr.shape)
    # look at the profile, despite setting nodata=0, there still isn't a nodata
value
    # this is because we need to update the profile and write out a new image with
    #  nodata set
    profile = src.profile
    print(profile)
```

```
Array shape: (3, 1860, 2041)
{'driver': 'GTiff', 'dtype': 'uint16', 'nodata': None, 'width': 2041, 'height':
1860, 'count': 3, 'crs': CRS.from_epsg(32621), 'transform': Affine(30.0, 0.0,
717345.0,
       0.0, -30.0, -2776995.0), 'blockxsize': 256, 'blockysize': 256, 'tiled':
True, 'compress': 'lzw', 'interleave': 'pixel'}
```

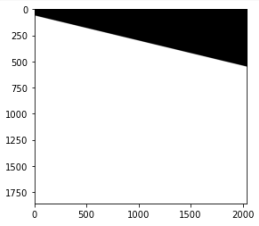
> **Note**
>
> We use `src.read([3, 2, 1])` in order to reverse the read order of our bands. With this the original blue, green, red order is read in reverse as red, green, blue, which is required for true color images.

To see what it looks like, let's use rasterio's `show` function which *sort of helps* when viewing multiband imagery.

```
from rasterio.plot import show
print(arr)
print('--------------')
show(arr)
```

```
[[[   0    0    0 ...    0    0    0]
  [   0    0    0 ...    0    0    0]
  [   0    0    0 ...    0    0    0]
  ...
  [6672 6168 6100 ... 6028 6015 6008]
  [6291 6457 6423 ... 6021 6001 6017]
  [6181 6727 6747 ... 6098 6023 5991]]

 [[   0    0    0 ...    0    0    0]
  [   0    0    0 ...    0    0    0]
  [   0    0    0 ...    0    0    0]
  ...
  [7083 6794 6731 ... 6760 6724 6685]
  [6951 7079 7016 ... 6699 6687 6715]
  [6864 7286 7338 ... 6817 6732 6659]]

 [[   0    0    0 ...    0    0    0]
  [   0    0    0 ...    0    0    0]
  [   0    0    0 ...    0    0    0]
  ...
  [7692 7518 7513 ... 7440 7432 7415]
  [7586 7590 7610 ... 7440 7411 7425]
  [7576 7743 7770 ... 7464 7443 7406]]]
--------------
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats
or [0..255] for integers).
```




```
<AxesSubplot:>
```

Arg… it looks terrible. A few things, the primary problem is that the zeros aren't being treated as missing data, and therefor messing everything up. Second, there is a message at the top of the image saying its `clipping... [0..255] for integers)`.

Let's check what the range of values are in our raster. To do this we will use `scipy stats`, and `.ravel()` to convert our `n` dimension array to a `1d` array.

```
from scipy import stats

print(stats.describe(arr.ravel()))
```

```
DescribeResult(nobs=11388780, minmax=(0, 24147), mean=6095.312502656123,
variance=7725568.74402071, skewness=-1.6071105370358574,
kurtosis=0.9303298625510847)
```

Since we currently have integer data that ranges from 0 to 24147, we should try scaling it, if we go to the documentation we would see that we need to divide all values by 100.

```
scaled_arr = arr / 100
print(scaled_arr.dtype)
```

```
float64
```

Then let's write the data back out again with an updated profile. Notice that the datatype (`dtype`) has changed since we divided by 100.

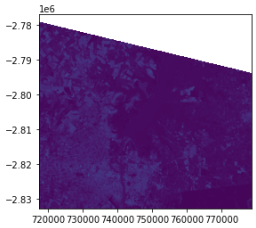
```
# update the profile for the new raster
profile = src.profile
profile.update(nodata=0, compress="lzw", dtype=scaled_arr.dtype)

# write out raster as RGB
with rasterio.open(
    "../temp/LS_scaled_20200518.tif",
    mode="w",
    **profile,  # unpack the profile arguments set above
) as new_dataset:
    new_dataset.write(scaled_arr, [1, 2, 3])
```

Note

Since the bands have already been put in the right order in the array (ie. RGB), we can just write it out as follows `.write(scaled_arr, [1, 2, 3])`.

There you go! A fixed raster. Try opening it in Qgis to make sure. We can look at it now using rasterio `show`, but again it isn't great, but better.

```
with rasterio.open("../temp/LS_scaled_20200518.tif", mode="r") as src:
    show(src,adjust='linear')
```



To help make all of this easier and more intuitive we will be presenting the use of `geowombat` for remote sensing applications later, start here.

Just as a preview, here's how to do this in `geowombat`.

```
import geowombat as gw
import matplotlib.pyplot as plt

fig, ax = plt.subplots(dpi=200)

# tell gw to read a blue green red
with gw.config.update(sensor="bgr"):
    with gw.open("../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF") as src:

        # see that bands names, blue green red are assigned
        print(src.band)

        # remove 0 value, rearrange band order
        temp = src.where(src != 0).sel(band=["red", "green", "blue"])

        # plot
        temp.gw.imshow(robust=True, ax=ax)

        #save to file
        temp.gw.to_raster(
            "../temp/LS_scaled_20200518.tif", verbose=0, n_workers=4,
overwrite=True
        )
```

```
<xarray.DataArray 'band' (band: 3)>
array(['blue', 'green', 'red'], dtype='<U5')
Coordinates:
  * band     (band) <U5 'blue' 'green' 'red'
```

```
  0%|          | 0/50 [00:00<?, ?it/s]
```

```
 18%|█▊        | 9/50 [00:00<00:00, 82.56it/s]
```

```
 36%|███▌      | 18/50 [00:00<00:00, 79.51it/s]
```

```
 52%|█████▏    | 26/50 [00:00<00:00, 78.19it/s]
```

```
 68%|██████▊   | 34/50 [00:00<00:00, 74.41it/s]
```

```
 84%|████████▍ | 42/50 [00:00<00:00, 75.57it/s]
```

```
100%|██████████| 50/50 [00:00<00:00, 76.03it/s]
```

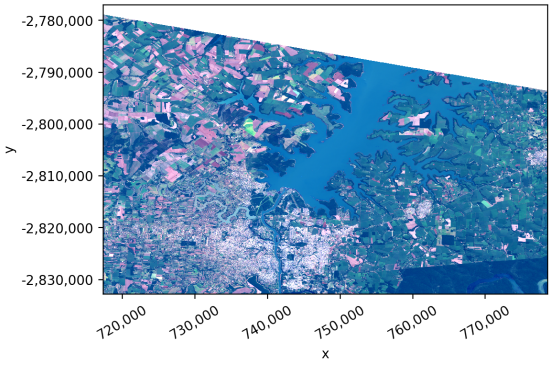
```
100%|██████████| 50/50 [00:00<00:00, 76.54it/s]
```

```
  0%|          | 0/14 [00:00<?, ?it/s]
```

```
 57%|█████▋    | 8/14 [00:00<00:00, 75.63it/s]
```

```
100%|██████████| 14/14 [00:00<00:00, 80.66it/s]
```

```
```



Credits: rasterio quickstart

Learning Objectives

- Reproject a raster with rasterio
- Reproject a raster with geowombat

Review

- Affine transformation
- Raster Coordinate Reference Systems

Reproject Rasters w. Rasterio and Geowombat

Reprojecting a Raster with Geowombat

Far and away the easiest way to handle raster data is by using geowombat. Here's an example of quickly and easily reprojecting a three band landsat image, and writing it to disk.

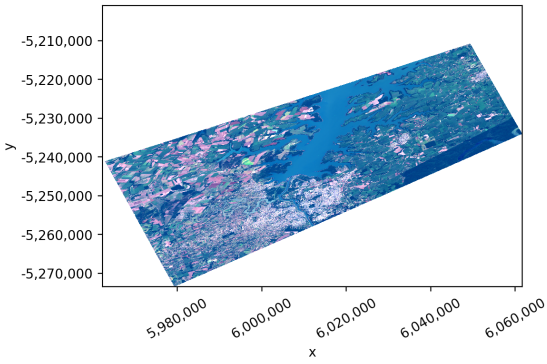
In order to reproject on the fly we are going to open the raster using `gw.config.update()`. The configuration manager allows easy control over opened raster dimensions, alignment, and transformations. All we need to do is pass a `ref_crs` to the configuration manager. We can also use the `resampling` method when we `open` the image, by default it will be `nearest`, but you can also choose one of ['average', 'bilinear', 'cubic', 'cubic_spline', 'gauss', 'lanczos', 'max', 'med', 'min', 'mode', 'nearest'].

```python
import geowombat as gw

proj4 = "+proj=aea +lat_1=20 +lat_2=60 +lat_0=40 +lon_0=-96 +x_0=0 +y_0=0
+datum=NAD83 +units=m +no_defs"
image = "../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF"

with gw.config.update(ref_crs=proj4):
    with gw.open(image, resampling="nearest") as src:

        src.gw.to_raster(
            "../temp/LC08_20200518_aea.tif",
            verbose=0,
            n_workers=4,  # number of process workers
            n_threads=2,  # number of thread workers ``dask.compute``
            overwrite=True,
        )
```

```
  0%|          | 0/130 [00:00<?, ?it/s]
  1%|          | 1/130 [00:00<00:16,  7.78it/s]
  5%|▍         | 6/130 [00:00<00:04, 27.91it/s]
  8%|▊         | 11/130 [00:00<00:03, 31.87it/s]
 13%|█▏        | 17/130 [00:00<00:02, 41.15it/s]
 17%|█▋        | 22/130 [00:00<00:03, 33.56it/s]
 20%|██        | 26/130 [00:00<00:02, 35.01it/s]
 24%|██▍       | 31/130 [00:00<00:02, 35.96it/s]
 27%|██▋       | 35/130 [00:01<00:03, 30.97it/s]
 30%|███       | 39/130 [00:01<00:02, 31.57it/s]
 34%|███▍      | 44/130 [00:01<00:02, 30.00it/s]
 37%|███▋      | 48/130 [00:01<00:03, 24.84it/s]
 39%|███▉      | 51/130 [00:01<00:03, 25.35it/s]
 42%|████▏     | 54/130 [00:01<00:03, 25.24it/s]
 44%|████▍     | 57/130 [00:02<00:03, 21.64it/s]
 46%|████▌     | 60/130 [00:02<00:03, 21.94it/s]
 48%|████▊     | 63/130 [00:02<00:03, 20.75it/s]
 51%|█████     | 66/130 [00:02<00:03, 19.31it/s]
 53%|█████▎    | 69/130 [00:02<00:02, 21.01it/s]
 55%|█████▌    | 72/130 [00:02<00:03, 18.28it/s]
 57%|█████▋    | 74/130 [00:02<00:03, 17.82it/s]
 61%|██████    | 79/130 [00:03<00:02, 22.24it/s]
 63%|██████▎   | 82/130 [00:03<00:02, 20.21it/s]
 65%|██████▌   | 85/130 [00:03<00:02, 21.08it/s]
 69%|██████▉   | 90/130 [00:03<00:01, 25.16it/s]
 72%|███████▏  | 93/130 [00:03<00:01, 26.23it/s]
 74%|███████▍  | 96/130 [00:03<00:01, 19.07it/s]
 79%|███████▉  | 103/130 [00:04<00:00, 28.55it/s]
 82%|████████▏ | 107/130 [00:04<00:00, 26.00it/s]
 86%|████████▌ | 112/130 [00:04<00:00, 29.47it/s]
 89%|████████▉ | 116/130 [00:04<00:00, 31.71it/s]
 92%|█████████▏| 120/130 [00:04<00:00, 31.86it/s]
 97%|█████████▋| 126/130 [00:04<00:00, 36.98it/s]
100%|██████████| 130/130 [00:04<00:00, 27.11it/s]
```

Let's take a look, remember from earlier that this image is stored as green, blue, red (rather than red, green, blue), so we will use `.sel(band=[3,2,1])` to put them back in the right order.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(dpi=200)

image = "../temp/LC08_20200518_aea.tif"
with gw.open(image) as src:
    src.where(src != 0).sel(band=[3, 2, 1]).gw.imshow(robust=True, ax=ax)
    plt.tight_layout(pad=1)
```



Too easy? Want something more complex? Try the same thing with Rasterio. Yes, there will be a little matrix algebra.

Calculate_default_transform Explained

How do we reproject a raster? Before we get into it, we need to talk some more… about `calculate_default_transform`. `calculate_default_transform` allows us to generate the transform matrix required for the new reprojected raster based on the characteristics of the original and the desired output CRS. Note that the `source` (src) is the original input raster, and the `destination` (dst) is the outputed reprojected raster.

First, remember that the transform matrix takes the following form (review affine transforms here):

$$\text{Transform} = \begin{bmatrix} xres & 0 & \Delta x \\ 0 & yres & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

Now let's calculate the tranform matrix for the destination raster:

```
import numpy as np
import rasterio
from rasterio.warp import reproject, Resampling, calculate_default_transform

dst_crs = "EPSG:3857"  # web mercator(ie google maps)

with rasterio.open("../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF") as src:

    # transform for input raster
    src_transform = src.transform

    # calculate the transform matrix for the output
    dst_transform, width, height = calculate_default_transform(
        src.crs,    # source CRS
        dst_crs,    # destination CRS
        src.width,    # column count
        src.height,  # row count
        *src.bounds,  # unpacks outer boundaries (left, bottom, right, top)
    )

print("Source Transform:\n",src_transform,'\n')
print("Destination Transform:\n", dst_transform)
```

```
Source Transform:
 | 30.00, 0.00, 717345.00|
| 0.00,-30.00,-2776995.00|
| 0.00, 0.00, 1.00|

Destination Transform:
 | 33.24, 0.00,-6105300.09|
| 0.00,-33.24,-2885952.71|
| 0.00, 0.00, 1.00|
```

Notice that in order to keep the same number of rows and columns that the resolution of the destination raster increased from 30 meters to 33.24 meters. Also the coordinates of the upper left hand corner have shifted (check $\Delta x, \Delta x$).

Reprojecting a Raster with Rasterio

Ok finally!

```
dst_crs = "EPSG:3857"  # web mercator(ie google maps)

with rasterio.open("../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF") as src:
    src_transform = src.transform

    # calculate the transform matrix for the output
    dst_transform, width, height = calculate_default_transform(
        src.crs,
        dst_crs,
        src.width,
        src.height,
        *src.bounds,  # unpacks outer boundaries (left, bottom, right, top)
    )

    # set properties for output
    dst_kwargs = src.meta.copy()
    dst_kwargs.update(
        {
            "crs": dst_crs,
            "transform": dst_transform,
            "width": width,
            "height": height,
            "nodata": 0,  # replace 0 with np.nan
        }
    )

    with rasterio.open("../temp/LC08_20200518_webMC.tif", "w", **dst_kwargs) as
dst:
        # iterate through bands
        for i in range(1, src.count + 1):
            reproject(
                source=rasterio.band(src, i),
                destination=rasterio.band(dst, i),
                src_transform=src.transform,
                src_crs=src.crs,
                dst_transform=dst_transform,
                dst_crs=dst_crs,
                resampling=Resampling.nearest,
            )
```

Fig. 54 Reprojected Landsat Image

Learning Objectives

- Resample a raster with rasterio & geowombat
- Learn how to match the extent

Review

- Affine transformation
- Raster Coordinate Reference Systems
- Reprojecting Rasters
- Interpolation Methods Explained

Resampling & Registering Rasters w. Rasterio and Geowombat

Why is Resampling Important?

Before you begin any analysis using rater data it is critical that you have rasters that are “co-registered”. Co-registration requires that any two rasters have the same resolution, and orientation - the origins can differ but they much align. *In other words, co-registration requires that cell centroids align perfectly for all intersecting areas.* Resampling is also extremely common during reprojection operations as it often requires changing the orientation, scale or resolution of an image.

Examples of data that is *not* co-registered

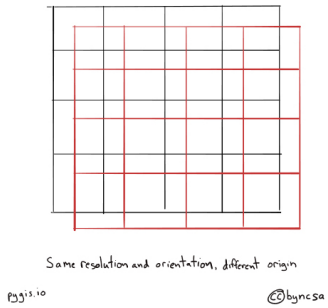


Fig. 55 Resampling rasters - different orientation

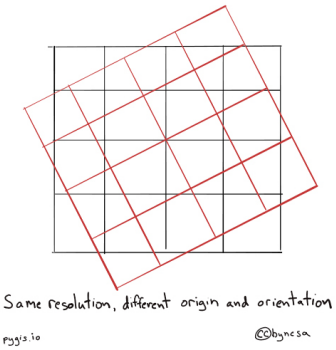


Fig. 56 Resampling rasters - different orientation and origin

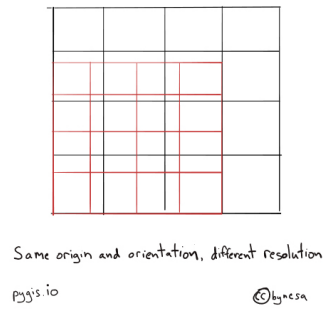


Fig. 57 Resampling rasters - different resolution

We can co-register images by resampling, and often reprojecting, one image to match another.

Methods for Resampling Explained

There are a number of methods to resample data, but they often take the form of “nearest neighbor”, “bilinear”, and “cubic convolusion” - these interpolation methods are explained here in some detail. But there are a number of other including: [‘average’, ‘cubic_spline’, ‘gauss’, ‘lanczos’, ‘max’, ‘med’, ‘min’, ‘mode’, ‘nearest’].

resampling direction

- Upsampling - converting to higher resolution/smaller cells.
- Downsampling - converting to lower resolution/larger cell sizes.

Simple Up/Downsampling in Rasterio & Geowombat

Rasterio Upsampling Example

Occationally you will need to resample your data by some factor, for instance you might want data upsampled to a courser resolution due to memory constraints.

Here’s an example of how to generate the `data` array and the `transform` needed to write it out. We will start by simply reading in the data and coersing a higher resolution, by adding more rows and columns. To understand the `transform`, let’s review affine transformations. Here we will update the scale values with the new resolution using S_y and S_x show below

$$\text{Scale transform: } \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
import rasterio
from rasterio.enums import Resampling

image = "../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF"

upscale_factor = 2

with rasterio.open(image) as dataset:

    # resample data to target shape using upscale_factor
    data = dataset.read(
        out_shape=(
            dataset.count,
            int(dataset.height * upscale_factor),
            int(dataset.width * upscale_factor)
        ),
        resampling=Resampling.bilinear
    )

    print('Shape before resample:', dataset.shape)
    print('Shape after resample:', data.shape[1:])

    # scale image transform
    dst_transform = dataset.transform * dataset.transform.scale(
        (dataset.width / data.shape[-1]),
        (dataset.height / data.shape[-2])
    )

    print('Transform before resample:\n', dataset.transform, '\n')
    print('Transform after resample:\n', dst_transform)

    ## Write outputs
    # set properties for output
    # dst_kwargs = src.meta.copy()
    # dst_kwargs.update(
    #     {
    #         "crs": dst_crs,
    #         "transform": dst_transform,
    #         "width": data.shape[-1],
    #         "height": data.shape[-2],
    #         "nodata": 0,
    #     }
    # )

    # with rasterio.open("../temp/LC08_20200518_15m.tif", "w", **dst_kwargs) as
dst:
    #     # iterate through bands
    #     for i in range(data.shape[0]):
    #           dst.write(data[i].astype(rasterio.uint32), i+1)
```

```
Shape before resample: (1860, 2041)
Shape after resample: (3720, 4082)
Transform before resample:
 | 30.00, 0.00, 717345.00|
| 0.00,-30.00,-2776995.00|
| 0.00, 0.00, 1.00|

Transform after resample:
 | 15.00, 0.00, 717345.00|
| 0.00,-15.00,-2776995.00|
| 0.00, 0.00, 1.00|
```

> **Reminder**
>
> Read up on affine transformations to help you understand the `transform.scale` function above.

Geowombat Up/Down Sampling Example

As always the easiest way to deal with resampling is by deploying geowombat. It's like a swiss army knife for kicking raster butt. Here we just need to set the desired resolution with `ref_res`, and the `resampling` method in the open statement. We have a number of resampling methods available depending on the context listed above. Writing a file is a bit more intuitive too.

```
import geowombat as gw

image = "../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF"

with gw.config.update(ref_res=15):
    with gw.open(image, resampling="bilinear") as src:
        print(src)

        # to write out simply:
        # src.gw.to_raster(
        #     "../temp/LC08_20200518_15m.tif",
        #     overwrite=True,
        # )
```

```
<xarray.DataArray (band: 3, y: 3720, x: 4082)>
dask.array<open_rasterio-5d3779c5d83524d14a4bbcf537a1f23d<this-array>, shape=(3,
3720, 4082), dtype=uint16, chunksize=(1, 256, 256), chunktype=numpy.ndarray>
Coordinates:
  * band     (band) int64 1 2 3
  * y        (y) float64 -2.777e+06 -2.777e+06 ... -2.833e+06 -2.833e+06
  * x        (x) float64 7.174e+05 7.174e+05 7.174e+05 ... 7.786e+05 7.786e+05
Attributes:
    transform:          (15.0, 0.0, 717345.0, 0.0, -15.0, -2776995.0)
    crs:                +init=epsg:32621
    res:                (15.0, 15.0)
    is_tiled:           1
    nodatavals:         (0, 0, 0)
    scales:             (1.0, 1.0, 1.0)
    offsets:            (0.0, 0.0, 0.0)
    filename:           ../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF
    resampling:         bilinear
    AREA_OR_POINT:      Area
    data_are_separate:  0
    data_are_stacked:   0
```

Much…. easier….

Co-registering Rasters (Aligning cells)

One common problem is how to get raster data to align on a cell by cell basis so you can complete your analysis. We will walk through how to do this for both rasterio and geowombat. In the following example we will learn how to align rasters with different origins, resolutions, or orientations.

Our example data will look at registering LandSat data with precipitation data from CHIRPS.

Example of Co-registering Rasters with Rasterio

Co-registering data is a bit complicated with rasterio, you simply need to choose an “reference image” to match the bounds, CRS, and cell size.

The original input data LandSat data is 30 meters and will be downsampled to match `precip` which is 500m. Also note the use of `nodata` to avoid missing values stored as 0. Note we can choose a number of `resampling` techniques listed above. For more detail on resampling go to Introduction to Raster CRS.

```
from rasterio.warp import reproject, Resampling, calculate_default_transform
import rasterio
def reproj_match(infile, match, outfile):
    """Reproject a file to match the shape and projection of existing raster.

    Parameters
    ----------
    infile : (string) path to input file to reproject
    match : (string) path to raster with desired shape and projection
    outfile : (string) path to output file tif
    """
    # open input
    with rasterio.open(infile) as src:
        src_transform = src.transform

        # open input to match
        with rasterio.open(match) as match:
            dst_crs = match.crs

            # calculate the output transform matrix
            dst_transform, dst_width, dst_height = calculate_default_transform(
                src.crs,     # input CRS
                dst_crs,     # output CRS
                match.width,   # input width
                match.height,  # input height
                *match.bounds,  # unpacks input outer boundaries (left, bottom,
right, top)
            )

        # set properties for output
        dst_kwargs = src.meta.copy()
        dst_kwargs.update({"crs": dst_crs,
                           "transform": dst_transform,
                           "width": dst_width,
                           "height": dst_height,
                           "nodata": 0})
        print("Coregistered to shape:", dst_height,dst_width,'\n
Affine',dst_transform)
        # open output
        with rasterio.open(outfile, "w", **dst_kwargs) as dst:
            # iterate through bands and write using reproject function
            for i in range(1, src.count + 1):
                reproject(
                    source=rasterio.band(src, i),
                    destination=rasterio.band(dst, i),
                    src_transform=src.transform,
                    src_crs=src.crs,
                    dst_transform=dst_transform,
                    dst_crs=dst_crs,
                    resampling=Resampling.nearest)
```

Now we can execute out code to co-register two rasters

```
LS = "../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF"
precip = "../data/precipitation_20200601_500m.tif"

# co-register LS to match precip raster
reproj_match(infile = LS,
             match= precip,
             outfile = '../temp/LS_reg_precip.tif')
```

```
Coregistered to shape: 136 157
 Affine | 0.00, 0.00,-61.49|
| 0.00,-0.00,-0.00|
| 0.00, 0.00, 1.00|
```

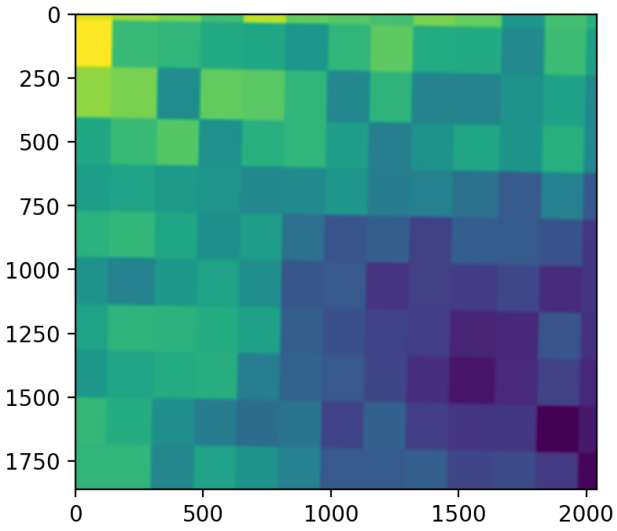
Example of Co-registering Rasters with Geowombat

Co-registering data is simple in geowombat, you simply need to choose an “reference image” to match the bounds, CRS, and cell size.

The original input data `precip` is currently 500m but will be upsampled to 30m as seen in `print(src)`. Also note the use of `nodata` to avoid missing values stored as -9999. Note we can choose a number of `resampling` techniques listed above. For more detail on resampling go to Introduction to Raster CRS.

```
import geowombat as gw
import matplotlib.pyplot as plt
fig, ax = plt.subplots(dpi=200)


LS = "../data/LC08_L1TP_224078_20200518_20200518_01_RT.TIF"
precip = "../data/precipitation_20200601_500m.tif"

with gw.config.update(ref_image=LS):
    with gw.open(precip, resampling="bilinear", nodata=-9999) as src:
        print(src)
        ax.imshow(src.data[0])

        # to write out simply:
        # src.gw.to_raster(
        #      "../temp/precip_20200601_30m.tif",
        #     overwrite=True,
        # )
```

```
<xarray.DataArray (band: 1, y: 1860, x: 2041)>
dask.array<open_rasterio-5ea1b887333ee835e34ed81f251d6116<this-array>, shape=(1,
1860, 2041), dtype=float32, chunksize=(1, 135, 157), chunktype=numpy.ndarray>
Coordinates:
  * band     (band) int64 1
  * y        (y) float64 -2.777e+06 -2.777e+06 ... -2.833e+06 -2.833e+06
  * x        (x) float64 7.174e+05 7.174e+05 7.174e+05 ... 7.785e+05 7.786e+05
Attributes: (12/13)
    transform:          (30.0, 0.0, 717345.0, 0.0, -30.0, -2776995.0)
    crs:                +init=epsg:32621
    res:                (30.0, 30.0)
    is_tiled:           1
    nodatavals:         (0,)
    scales:             (1.0,)
    ...                 ...
    descriptions:       ('precipitation',)
    filename:           ../data/precipitation_20200601_500m.tif
    resampling:         bilinear
    AREA_OR_POINT:      Area
    data_are_separate:  0
    data_are_stacked:   0
```



Although it's a bit hard to tell, the cell size is now 30m (see the slices of the larger original cells at the top of the image).

> **Note**
>
> Also note that geowombat's built in `.imshow` functionality doesn't currently work for single band images. To get an image we have to extract a `np.ndarray` that has the shape `(1, 1860, 2041)`. Where `(1,.,.)` is the band, so to get a 2d array we select `data[0]`. We also avoid missing values (-9999).

Band Math w. Rasterio

> **Learning Objectives**
>
> - Conduct mathematical operations on raster bands with rasterio
> - Understand the requirements for successful mathematical operations

> **Review**
>
> - Reproject Rasters w. Rasterio and Geowombat
> - Resampling Rasters w. Rasterio and Geowombat

Band math is useful when you want to perform a mathematical operation to each pixel value in a raster. You might find band math helpful for calculating NDVI or multiplying all values by a constant.

Setup

To begin, we will import our modules (click the + below to show code cell).

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt
import rasterio
from rasterio.transform import Affine
```

Band Math with rasterio with multiple images

`Rasterio` makes band math relatively straightforward since the rasters are essentially read in as numpy arrays, so you can perform math on the raster arrays just like any numpy array.

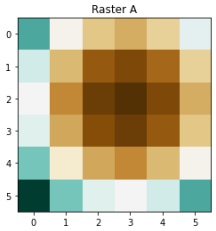
> **Attention**
>
> Mathematical operations on rasters using `rasterio` are not spatially aware. Any mathematical operation with multiple rasters will work even if the rasters are not representing the same geographical extent. Consequently, you need to ensure that the cell size and extent represented in all rasters are the same. In other words, if you are using two rasters in a mathematical operation, they must have the same shape (same number of rows and columns).

In this example we will write two raster files to the disk: `math_raster_a.tif` and `math_raster_b.tif`. We will then read then back in and do math on them.
Let's generate some rasters (click the + below to show code cell).

```
# Generate mesh grid for rasters
x = np.linspace(-90, 90, 6)
y = np.linspace(90, -90, 6)
X, Y = np.meshgrid(x, y)

# Generate values for mesh grid
Z1 = np.abs(((X - 10) ** 2 + (Y - 10) ** 2) / 1 ** 2)
Z2 = np.abs(((X + 10) ** 2 + (Y + 10) ** 2) / 2.5 ** 2)
Z3 = np.abs(((X + 3) + (Y - 8) ** 2) / 3 ** 2)

# Generate raster values for two rasters
Z_a = (Z1 - Z2)
Z_b = (Z2 - Z3)

# Set transform
xres = (x[-1] - x[0]) / len(x)
yres = (y[-1] - y[0]) / len(y)
transform = Affine.translation(x[0] - xres / 2, y[0] - yres / 2) *
Affine.scale(xres, yres)

# Save first raster
with rasterio.open(
        "../temp/math_raster_a.tif",
        mode="w",
        driver="GTiff",
        height=Z_a.shape[0],
        width=Z_a.shape[1],
        count=1,
        dtype=Z_a.dtype,
        crs="+proj=latlong",
        transform=transform,
) as new_dataset:
        new_dataset.write(Z_a, 1)

# Save second raster
with rasterio.open(
        "../temp/math_raster_b.tif",
        mode="w",
        driver="GTiff",
        height=Z_b.shape[0],
        width=Z_b.shape[1],
        count=1,
        dtype=Z_b.dtype,
        crs="+proj=latlong",
        transform=transform,
) as new_dataset:
        new_dataset.write(Z_b, 1)
```

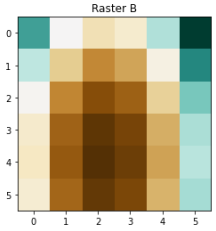
Next, we'll view the raster values.

```
# Open raster and plot
raster_a = rasterio.open("../temp/math_raster_a.tif").read(1)
plt.imshow(raster_a, cmap = "BrBG")
plt.title("Raster A")
plt.show()

# View raster values
print(raster_a)
```




```
[[13776.    8586.24  5573.76  4738.56  6080.64  9600.  ]
 [10256.64  5066.88  2054.4   1219.2   2561.28  6080.64]
 [ 8914.56  3724.8    712.32  -122.88  1219.2   4738.56]
 [ 9749.76  4560.    1547.52   712.32  2054.4   5573.76]
 [12762.24  7572.48  4560.    3724.8   5066.88  8586.24]
 [17952.   12762.24  9749.76  8914.56 10256.64 13776.  ]]
```

```
# Open raster and plot
raster_b = rasterio.open("../temp/math_raster_b.tif").read(1)
plt.imshow(raster_b, cmap = "BrBG")
plt.title("Raster B")
plt.show()

# View raster values
print(raster_b)
```

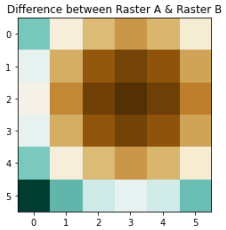



```
[[1886.55555556 1168.31555556  864.79555556  975.99555556 1501.91555556
  2442.55555556]
 [1453.91555556  735.67555556  432.15555556  543.35555556 1069.27555556
  2009.91555556]
 [1147.99555556  429.75555556  126.23555556  237.43555556  763.35555556
  1703.99555556]
 [ 968.79555556  250.55555556  -52.96444444   58.23555556  584.15555556
  1524.79555556]
 [ 916.31555556  198.07555556 -105.44444444    5.75555556  531.67555556
  1472.31555556]
 [ 990.55555556  272.31555556  -31.20444444   79.99555556  605.91555556
  1546.55555556]]
```

Example band math operations

We can get the difference between the two rasters.

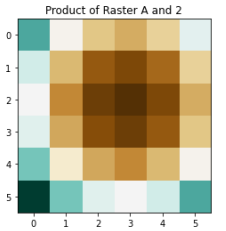
```
# Get difference
difference_a_b = raster_a - raster_b

# Plot raster
plt.imshow(difference_a_b, cmap = "BrBG")
plt.title("Difference between Raster A & Raster B")
plt.show()

# Show raster values
print("Raster values:\n", difference_a_b)
```




```
Raster values:
 [[11889.44444444  7417.92444444  4708.96444444  3762.56444444
   4578.72444444  7157.44444444]
 [ 8802.72444444  4331.20444444  1622.24444444   675.84444444
   1492.00444444  4070.72444444]
 [ 7766.56444444  3295.04444444   586.08444444  -360.31555556
    455.84444444  3034.56444444]
 [ 8780.96444444  4309.44444444  1600.48444444   654.08444444
   1470.24444444  4048.96444444]
 [11845.92444444  7374.40444444  4665.44444444  3719.04444444
   4535.20444444  7113.92444444]
 [16961.44444444 12489.92444444  9780.96444444  8834.56444444
   9650.72444444 12229.44444444]]
```

We can multiply a raster by a constant.

```
# Get product
product_a = raster_a * 2

# Plot raster
plt.imshow(product_a, cmap = "BrBG")
plt.title("Product of Raster A and 2")
plt.show()

# Show raster values
print("Raster values:\n", product_a)
```



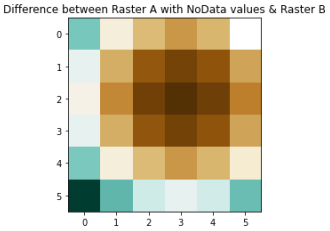

```
Raster values:
 [[27552.   17172.48 11147.52  9477.12 12161.28 19200.  ]
 [20513.28 10133.76  4108.8   2438.4   5122.56 12161.28]
 [17829.12  7449.6   1424.64  -245.76  2438.4   9477.12]
 [19499.52  9120.    3095.04  1424.64  4108.8  11147.52]
 [25524.48 15144.96  9120.    7449.6  10133.76 17172.48]
 [35904.   25524.48 19499.52 17829.12 20513.28 27552.  ]]
```

Band math with NoData values

If a pixel has a value of `nan`, `None`, or `NoData` value, those pixels will automatically be ignored in any band math. The output raster will maintain the `nan`, `None`, or `NoData` value at that pixel location.

Not all rasters, however, use those values to signify that a pixel has no value. Some rasters might use 0 or another number to indicate no value. In that case, we have to explicitly mark that pixel to be skipped.

```
# Create a copy of first raster
raster_0 = raster_a.copy()

# Set a pixel value to 0 as an example, which will signify NoData
# (top right pixel)
raster_0[0, 5] = 0

# Mask out any NoData (0) values
raster_0_masked = np.ma.masked_array(raster_0, mask = (raster_0 == 0))

# Get difference between masked raster and second raster
difference_0_b = raster_0_masked - raster_b

# Plot raster
plt.imshow(difference_0_b, cmap = "BrBG")
plt.title("Difference between Raster A with NoData values & Raster B")
plt.show()

# Show raster values
print("Raster values:\n", difference_0_b)
```




```
Raster values:
 [[11889.444444444445 7417.924444444445 4708.964444444445
  3762.564444444444 4578.724444444444 --]
 [8802.724444444444 4331.204444444445 1622.2444444444445
  675.8444444444445 1492.0044444444443 4070.7244444444436]
 [7766.564444444444 3295.044444444447 586.0844444444443
  -360.31555555555553 455.84444444444455 3034.564444444444]
 [8780.964444444444 4309.444444444444 1600.4844444444443
  654.0844444444443 1470.2444444444445 4048.964444444445]
 [11845.924444444445 7374.404444444444 4665.444444444444
  3719.0444444444447 4535.204444444445 7113.924444444445]
 [16961.444444444445 12489.924444444445 9780.964444444444
  8834.564444444444 9650.724444444444 12229.444444444445]]
```

Example: Calculating NDVI

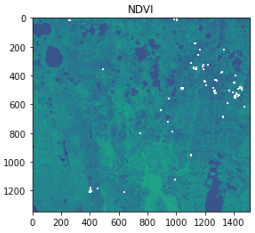
In the example below, we will read in a clipped Landsat 8, Collection 2 Level-2 image and use the band math concepts to calculate the normalized difference vegetation index (NDVI) for the image. As you may recall, NDVI is a spectral approach used to assess vegetation. The formula for NDVI is:

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

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

High NDVI values (towards 1) reflect a higher density of green vegetation, and low values (towards -1) reflect a lower density.

```
# Open raster (Landsat 8, Collection 2 Level-2)
# Band 1 - Blue, Band 2 - Green, Band 3 - Red, Band 4 - Near Infrared
# Source: https://www.usgs.gov/centers/eros/science/usgs-eros-archive-landsat-
archives-landsat-8-9-olitirs-collection-2-level-2
with rasterio.open("../data/LC08_L2SP_016040_20210317_20210328_02_T1_clip.tif",
mode = "r", nodata = 0) as src:

    # Get red band
    band_red = src.read(3)

    # Get NIR band
    band_nir = src.read(4)

    # Allow division by zero
    np.seterr(divide = "ignore", invalid = "ignore")

    # Calculate NDVI
    ndvi = (band_nir.astype(float) - band_red.astype(float)) / (band_nir +
band_red)

# Set pixels whose values are outside the NDVI range (-1, 1) to NaN
# Likely due to errors in the Landsat imagery
ndvi[ndvi > 1] = np.nan
ndvi[ndvi < -1] = np.nan

# Plot raster
plt.imshow(ndvi)
plt.title("NDVI")
plt.show()

# Show raster values
print("Raster values:\n", ndvi)
```




```
Raster values:
 [[0.1045768  0.08932138 0.09787864 ... 0.21016596 0.27657979 0.24717701]
 [0.07188325 0.1010677  0.1213656  ... 0.22981464 0.24469611 0.23835025]
 [0.03591049 0.14069341 0.17257186 ... 0.22467695 0.2150974  0.21510794]
 ...
 [0.27475696 0.24538079 0.24003341 ... 0.2692435  0.26424982 0.2754381 ]
 [0.28198954 0.26983464 0.25958875 ... 0.26693343 0.2685162  0.25831458]
 [0.27442722 0.28693113 0.2514506  ... 0.27998016 0.28904986 0.2745722 ]]
```

Band Math with GeoWombat

For band math with `GeoWombat`, see the chapter on Band Math & Vegetation Indices.

Replacing Values w. Rasterio

Learning Objectives

- Replace and interpolate values in a raster with rasterio

> **Review**
>
> - Spatial Raster Data

Imagery may sometimes have errors due to factors such as noise, distortion, or sensor errors. Some pixels may have extremely high or low values or no value at all. One way to resolve this issue is to manually replace a pixel value with another pixel value. Another option is to interpolate the pixel value based on the values of the pixel's neighbors.

We'll explore how to replace raster values with `rasterio`.

Setup

First, we will import our modules (click the + below to show code cell).

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt
import rasterio
from rasterio.transform import Affine
from rasterio.fill import fillnodata
```

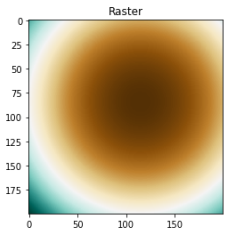
Next, we will generate a sample raster to be used (click the + below to show code cell).

```
# Generate mesh grid for rasters
x = np.linspace(-90, 90, 200)
y = np.linspace(90, -90, 200)
X, Y = np.meshgrid(x, y)

# Generate values for mesh grid
Z1 = np.abs(((X - 10) ** 2 + (Y - 10) ** 2) / 1 ** 2)
Z2 = np.abs(((X + 10) ** 2 + (Y + 10) ** 2) / 2.5 ** 2)

# Generate raster values
Z = (Z1 - Z2)

# Set transform
xres = (x[-1] - x[0]) / len(x)
yres = (y[-1] - y[0]) / len(y)
transform = Affine.translation(x[0] - xres / 2, y[0] - yres / 2) *
Affine.scale(xres, yres)

# Save raster
with rasterio.open(
        "../temp/replace_values_raster.tif",
        mode="w",
        driver="GTiff",
        height=Z.shape[0],
        width=Z.shape[1],
        count=1,
        dtype=Z.dtype,
        crs="+proj=latlong",
        transform=transform,
) as new_dataset:
        new_dataset.write(Z, 1)
```

Replace values with rasterio

We will open the example raster that we generated above.

```
# Open raster and plot
raster_file = rasterio.open("../temp/replace_values_raster.tif")
raster = raster_file.read(1)
plt.imshow(raster, cmap="BrBG")
plt.title("Raster")
plt.show()
```



Replace values with a specified number

Let's say that we want to change the pixel value at row 150, column 100 because it's wrong. We can simply call that pixel value by its row index and column index.

```
# Replace value with 0 at one location
raster[150, 100] = 0
raster[150, 100]
```

```
0.0
```

We can also change multiple pixel values by slicing. In this case, we replace the values in rows 99-101 and columns 6-8 with the value `0`.

```
# Replace values with 0 at multiple locations
raster[99:102, 6:9] = 0
raster[99:102, 6:9]
```

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Finally, we can change any pixel values that are of a certain value.

```
# Replace values with 0 if they are greater than or equal to certain number (in
this case, 13776)
raster[raster >= 13776] = 0
raster
```

```
array([[    0.        , 13618.93851165, 13463.25153405, ...,
         9371.19123254,  9484.9083609 ,  9600.        ],
       [13660.9083609 , 13503.84687255, 13348.15989495, ...,
         9256.09959344,  9369.8167218 ,  9484.9083609 ],
       [13547.19123254, 13390.1297442 , 13234.4427666 , ...,
         9142.38246509,  9256.09959344,  9371.19123254],
       ...,
       [    0.        ,     0.        ,     0.        , ...,
        13234.4427666 , 13348.15989495, 13463.25153405],
       [    0.        ,     0.        ,     0.        , ...,
        13390.1297442 , 13503.84687255, 13618.93851165],
       [    0.        ,     0.        ,     0.        , ...,
        13547.19123254, 13660.9083609 ,     0.        ]])
```

Replace values through interpolation

Sometimes, we don't know or have an exact value to replace pixel values with. We can "fill in" those pixel values through interpolation. Recall that interpolation uses the pixel values surrounding a certain pixel to determine the value for that certain pixel.

In the following example, we will interpolate the values for the pixels that were previously set to `0`.

`Rasterio` provides a function `fillnodata()` that does this for us. In addition to specifying a raster, we also need to provide a mask, which tells the function which pixel values need to be filled in. The mask can either be an array of Boolean values (`True` or `False`, where `False` indicates pixels to be filled in) or numbers (where values equal to `0` indicate pixels to be filled in and values equal to `1` indicate pixels to ignore).

For more information this function, see the function documentation.

> **Important**
>
> Mask must be in the same shape (number of rows and columns) as that of the input raster.

Below, we will interpolate the pixels whose values were previously set to `0`.

```
# Create a Boolean mask (True/False), with a value of False for pixels that equal
0
mask_boolean = (raster != 0)
mask_boolean
```

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

```
# Create a value mask, with a value of 0 for pixels that equal 0
mask_numbers = np.zeros_like(raster)
mask_numbers[raster > 0] = 255
mask_numbers
```

```
array([[  0., 255., 255., ..., 255., 255., 255.],
       [255., 255., 255., ..., 255., 255., 255.],
       [255., 255., 255., ..., 255., 255., 255.],
       ...,
       [  0.,   0.,   0., ..., 255., 255., 255.],
       [  0.,   0.,   0., ..., 255., 255., 255.],
       [  0.,   0.,   0., ..., 255., 255.,   0.]])
```

```
# Fill in missing values with interpolation
# Can use either a Boolean mask or a value mask
fillnodata(raster, mask = mask_boolean, max_search_distance = 1000)
```

```
array([[13604.37988281, 13618.93847656, 13463.25195312, ...,
         9371.19140625,  9484.90820312,  9600.        ],
       [13660.90820312, 13503.84667969, 13348.16015625, ...,
         9256.09960938,  9369.81640625,  9484.90820312],
       [13547.19140625, 13390.12988281, 13234.44238281, ...,
         9142.3828125 ,  9256.09960938,  9371.19140625],
       ...,
       [13687.77148438, 13674.72460938, 13698.35644531, ...,
        13234.44238281, 13348.16015625, 13463.25195312],
       [13698.04101562, 13686.17285156, 13709.640625  , ...,
        13390.12988281, 13503.84667969, 13618.93847656],
       [13688.7734375 , 13672.33984375, 13704.9375    , ...,
        13547.19140625, 13660.90820312, 13618.93847656]])
```

Finally, we can check the raster values to see the interpolated values.

```
# Print raster array
raster
```

```
array([[13604.37988281, 13618.93847656, 13463.25195312, ...,
         9371.19140625,  9484.90820312,  9600.        ],
       [13660.90820312, 13503.84667969, 13348.16015625, ...,
         9256.09960938,  9369.81640625,  9484.90820312],
       [13547.19140625, 13390.12988281, 13234.44238281, ...,
         9142.3828125 ,  9256.09960938,  9371.19140625],
       ...,
       [13687.77148438, 13674.72460938, 13698.35644531, ...,
        13234.44238281, 13348.16015625, 13463.25195312],
       [13698.04101562, 13686.17285156, 13709.640625  , ...,
        13390.12988281, 13503.84667969, 13618.93847656],
       [13688.7734375 , 13672.33984375, 13704.9375    , ...,
        13547.19140625, 13660.90820312, 13618.93847656]])
```

```
# Print subset of raster around row 150, column 100
raster[148:153, 98:103]
```

```
array([[2835.38891602, 2813.02954102, 2792.04467773, 2772.43432617,
        2754.19824219],
       [2923.72485352, 2901.36547852, 2880.38061523, 2860.77026367,
        2842.53417969],
       [3013.43530273, 2991.07592773, 2978.80517578, 2950.48071289,
        2932.24462891],
       [3104.52050781, 3082.16088867, 3061.17602539, 3041.56567383,
        3023.32983398],
       [3196.97998047, 3174.62036133, 3153.63549805, 3134.02514648,
        3115.78930664]])
```

```
# Print subset of raster around rows 99-101, columns 6-8]
raster[97:104, 4:11]
```

```
array([[8391.83398438, 8240.27050781, 8090.08154297, 7941.26757812,
        7793.82763672, 7647.76220703, 7503.07128906],
       [8410.0703125 , 8258.50683594, 8108.31787109, 7959.50341797,
        7812.06347656, 7665.99804688, 7521.30712891],
       [8429.68066406, 8278.1171875 , 8072.66064453, 7934.33056641,
        7755.44726562, 7685.60839844, 7540.91748047],
       [8450.66503906, 8299.1015625 , 8203.32421875, 7863.87109375,
        7767.40527344, 7706.59326172, 7561.90234375],
       [8473.02441406, 8321.4609375 , 8150.89306641, 7999.92822266,
        7799.43115234, 7728.95263672, 7584.26171875],
       [8496.75878906, 8345.1953125 , 8195.00683594, 8046.19189453,
        7898.75195312, 7752.68652344, 7607.99560547],
       [8521.8671875 , 8370.30371094, 8220.11523438, 8071.30029297,
        7923.86035156, 7777.79492188, 7633.10400391]])
```

Replace values with `GeoWombat`

For replacing raster values with `GeoWombat`, see the chapter on Editing Rasters and Remotely Sensed Data.

Rasterize Vectors w. Rasterio

Learning Objectives

- Convert vector data into raster format with rasterio
- Understand the requirements for successful conversion

Review

- Spatial Vector Data
- Spatial Raster Data
- Reproject Rasters w. Rasterio and Geowombat

Rasterizing vectors can be helpful if you want to incorporate vector data (i.e., point, line, or polygon) in your raster analysis. The process is essentially what the name suggests: We take a vector and convert it into pixels. This can be done with `rasterio`.

Setup

We'll begin by importing our modules (click the + below to show code cell).

```
# Import modules
import geopandas as gpd
import matplotlib.pyplot as plt
import rasterio
from rasterio import features
from rasterio.enums import MergeAlg
from rasterio.plot import show
import numpy as np
```

Rasterize vectors with rasterio

We'll read in the vector file of some of California's counties. We will also read in a raster file to get the raster's metadata (i.e., coordinate system) so that we can apply those parameters to the vector file. In other words, the raster will serve as a "reference" for the the rasterization of the vector. In particular, we are going to match the shape (number of rows and columns) and the transform (UL corner location, cell size etc). For a refresher on transforms, please see the chapter on Affine Transforms.

Important

The vector and raster **must be** be in the same coordinate system. If not, you'll need to re-project one of them so they are the same. To re-project vectors, see the chapter on Understanding CRS Codes. To re-project rasters, see the chapter on Reproject Rasters w. Rasterio and Geowombat.

One important parameter in this function is `all_touched` which determines how zones are determined by polygons relative to the reference raster's cell centroids. Setting it to `False` implies that membership in a zone, defined by a polygon geometry, should be defined by whether it contains the centroid of a cell. `True` includes any cell that geometry boundary intersects.

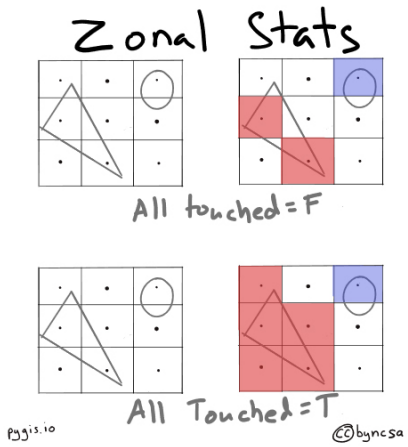


Fig. 58 all_touched determines the extent of zones

```
# Read in vector
vector =
gpd.read_file(r"../_static/e_vector_shapefiles/sf_bay_counties/sf_bay_counties.shp
")

# Get list of geometries for all features in vector file
geom = [shapes for shapes in vector.geometry]

# Open example raster
raster = rasterio.open(r"../_static/e_raster/bay-area-wells_kde_sklearn.tif")
```

Rasterize Binary Values for Shapes

With our data loaded, we can rasterize the vector using the metadata from the raster using `rasterize()` in the `rasterio.features` module. For more information on this function, check out the `rasterio` documentation.

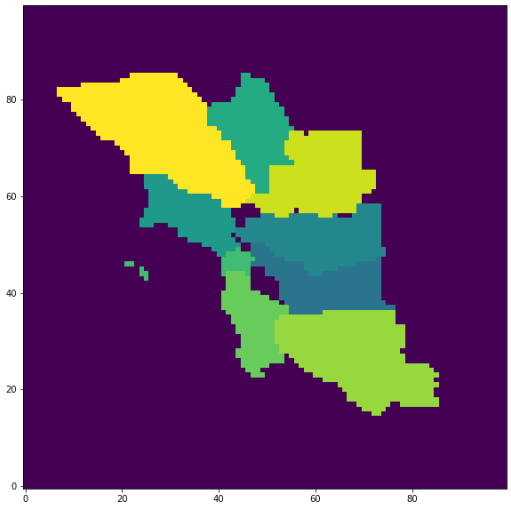
```
# Rasterize vector using the shape and coordinate system of the raster
rasterized = features.rasterize(geom,
                                out_shape = raster.shape,
                                fill = 0,
                                out = None,
                                transform = raster.transform,
                                all_touched = False,
                                default_value = 1,
                                dtype = None)

# Plot raster
fig, ax = plt.subplots(1, figsize = (10, 10))
show(rasterized, ax = ax)
plt.gca().invert_yaxis()
```

Rasterize Attribute Value using Rasterio

Often we want to burn in the value of a shapefile's attributes to the raster. We can do this by creating geometry, value pairs. In this example we take create a columns called `id` and assign the same values as the index. `id` will then be used to create our (geometry, value) pairs used for rasterization.

Note we use `all_touched=True` to avoid gaps between counties, which can introduce its own problems b/c two counties can compete for the same cell.

```
# create a numeric unique value for each row
vector['id'] = range(0,len(vector))

# create tuples of geometry, value pairs, where value is the attribute value you
want to burn
geom_value = ((geom,value) for geom, value in zip(vector.geometry, vector['id']))

# Rasterize vector using the shape and transform of the raster
rasterized = features.rasterize(geom_value,
                                out_shape = raster.shape,
                                transform = raster.transform,
                                all_touched = True,
                                fill = -5,   # background value
                                merge_alg = MergeAlg.replace,
                                dtype = np.int16)

# Plot raster
fig, ax = plt.subplots(1, figsize = (10, 10))
show(rasterized, ax = ax)
plt.gca().invert_yaxis()
```



Finally, we can save the rasterized vector out.

```
with rasterio.open(
        "../temp/rasterized_vector.tif", "w",
        driver = "GTiff",
        transform = raster.transform,
        dtype = rasterio.uint8,
        count = 1,
        width = raster.width,
        height = raster.height) as dst:
    dst.write(rasterized, indexes = 1)
```

Window Operations with Rasterio and GeoWombat

Learning Objectives

- Conduct and understand window operations with rasterio
- Conduct window operations with GeoWombat

Review

- Spatial Raster Data

Background

Moving windows or filters are often used in raster analysis. For example, they can be used to obtain the maximum value within a certain neighborhood, to smooth out values, or detect holes or edges (i.e., where pixel values that are near each other change abruptly).

These moving windows can also be called filters or kernels. They can be of many different sizes and shapes (the most common is 3-cell-by-3-cell rectangular window) and can have different or the same values for each cell. The center of the filter can be called the target cell or the center pixel, and the surrounding cells are referred to as the neighbors.

The filter passes through all *non-edge* cells in the raster. During each pass of the filter, the center cell is updated based on the cells adjacent to it. In the `3x3` filter example, the center cell is updated by the eight cells that neighbor it. The filter then pulls the values from the neighboring cells and the center pixel itself, performs a calculation based on the filter values (e.g., calculates the mean), reports that resulting value back to the identical location of the original pixel, moves to the next pixel, and repeats the process.

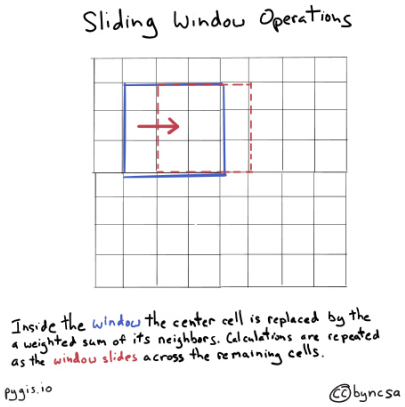


Fig. 59 Sliding window operations move across an entire raster.

The filter values can essentially be any number–they can be the same across the filter or be all different. The values determine how the output is calculated (equation below assuming a `3x3` filter for nine cells total in the kernel):

$$X_w = \sum_{i=1}^{9} X_i k_i$$

where: X_{i} = raster cell value

k_{i} = kernel cell value

i = index of cells in the nine kernel cell values

The values also determine *what* is calculated. For example, setting all filter values to `1` will result in the filter outputting the sum of all raster pixel values within the filter. Setting all filter values to `1/9` for a `3x3` filter will result in the filter outputting the average of all raster pixel values within the filter.

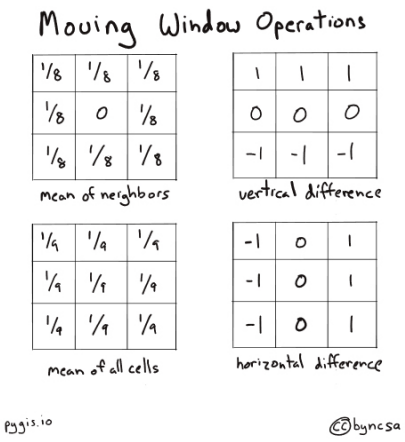


Fig. 60 Example filter values for `3x3` moving windows. Different filter value combinations and arrangements can produce different outputs.

For more information on moving windows, see the "Neighborhood Operations" section in this chapter on raster geoprocessing.

> **Attention**
>
> With window operations, the edge pixels will generally be cut out from the output because those edge pixels do not have neighboring pixels all around it. As window operations generally require a pixel to be surrounded on all four sides, we cannot perform calculations on these edge pixels. The pixels that constitute the "edge" depends on the shape of the kernel. For example, a `3x3` kernel only requires 1 neighboring pixel in each direction, so only the outer ring of pixels will be cut out from the output. A `5x5` kernel requires 2 neighboring pixels in each direction, so the two outer rings of pixels will be cut out.

Setup

We'll explore two methods, one using `rasterio` and another using `GeoWombat`.

First, we'll import our modules (click the + below to show code cell).

```
# Import modules
import geowombat as gw
import numpy as np
from itertools import product
import rasterio
from rasterio.transform import Affine
import matplotlib.pyplot as plt
```

Window operations with rasterio

The most intuitive way to perform window operations in Python is to use a `for` loop. With this method, we would iterate through each non-edge pixel, obtain the surrounding pixel values and the center pixel value, perform some sort of calculation, report that resulting value back to the identical location of the original pixel, move to the next pixel, and repeat the process. Iterating through each pixel, however, has the potential to be extremely time and computationally intensive (there could be many, many pixels).

To mitigate this limitation, instead of using a `for` loop, we can get each neighbor value simultaneously for all non-edge pixels. Another way to think of it is that instead of using a `for` loop to iterate through each pixel to get neighboring values, we can iterate through each kernel position to get the neighboring value corresponding to a certain kernel position for all non-edge pixels at once. This is what a vectorized sliding window does.

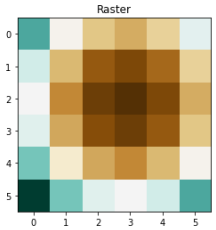
The vectorized sliding window is grounded in the concept that each position in the kernel has a certain relative position or offset (e.g., one pixel to the left) from the center pixel of the kernel. This method works because each pixel in a raster is in essence a neighbor to at least one other pixel. The vectorized sliding window simply offsets itself within the raster array extent so that each pixel that falls within the vectorized sliding window is the neighbor of the same relative position (e.g., one pixel to the left) to its center pixel.

A vectorized sliding window is created for each position in the kernel. The vectorized sliding window of a certain kernel position is applied over the raster and obtains–all at once–the neighboring pixel value corresponding to that kernel position for all non-edge pixels. The size of vectorized sliding window is dependent on the size of the raster and kernel but will always fall between the two.

For more conceptual information on vectorized sliding windows and how they compare to iterating through each pixel, see this article.

Let's create a raster (click the + below to show code cell).

```
# Generate mesh grid for rasters
x = np.linspace(-90, 90, 6)
y = np.linspace(90, -90, 6)
X, Y = np.meshgrid(x, y)

# Generate values for mesh grid
Z1 = np.abs(((X - 10) ** 2 + (Y - 10) ** 2) / 1 ** 2)
Z2 = np.abs(((X + 10) ** 2 + (Y + 10) ** 2) / 2.5 ** 2)
Z3 = np.abs(((X + 3) + (Y - 8) ** 2) / 3 ** 2)

# Generate raster values
Z = (Z1 - Z2)

# Set transform
xres = (x[-1] - x[0]) / len(x)
yres = (y[-1] - y[0]) / len(y)
transform = Affine.translation(x[0] - xres / 2, y[0] - yres / 2) *
Affine.scale(xres, yres)

# Save first raster
with rasterio.open(
        "../temp/window_raster.tif",
        mode="w",
        driver="GTiff",
        height=Z.shape[0],
        width=Z.shape[1],
        count=1,
        dtype=Z.dtype,
        crs="+proj=latlong",
        transform=transform,
) as new_dataset:
        new_dataset.write(Z, 1)

# Open and read raster
src = rasterio.open("../temp/window_raster.tif")
raster = src.read(1)

# Plot raster
plt.imshow(raster, cmap="BrBG")
plt.title("Raster")
plt.show()

# Show raster values
print(raster)
```




```
[[13776.    8586.24  5573.76  4738.56  6080.64  9600.  ]
 [10256.64  5066.88  2054.4   1219.2   2561.28  6080.64]
 [ 8914.56  3724.8    712.32  -122.88  1219.2   4738.56]
 [ 9749.76  4560.    1547.52   712.32  2054.4   5573.76]
 [12762.24  7572.48  4560.    3724.8   5066.88  8586.24]
 [17952.   12762.24  9749.76  8914.56 10256.64 13776.  ]]
```

Create kernel

Second, we can generate a kernel array. The array can consist of a single value or multiple values. Below, we generate a `3x3` kernel consisting of the value `1/9`. This kernel will output the average value of the center cell and its surrounding eight neighbors.

> **Important**
>
> The kernel should have an odd number of rows and columns and should not have more rows and columns than the input raster.

> **Tip**
>
> To create a non-rectangular shape, simply add `0`s in the kernel positions to be ignored.

```
# Create a kernel to calculate the average
kernel = np.full((3, 3), 1/9)

# Get kernel shape
kernel_shape = kernel.shape

# Convert the kernel to a flattened array
kernel_array = np.ravel(kernel)
```

Create output arrays

Next, we will create two arrays that will store our calculations.

The first array is the output array, which has the same shape as that of the input raster. We will initially fill the array with placeholder values. This array will be the final result that is exported and saved.

```
# Create raster array with placeholder values in shape of raster
output_rio = np.full((raster.shape[0], raster.shape[1]), -9999)

# Set array data type
output_rio = output_rio.astype(np.float64) ###

# Display raster array with placeholder values
print(output_rio)
```

```
[[-9999. -9999. -9999. -9999. -9999. -9999.]
 [-9999. -9999. -9999. -9999. -9999. -9999.]
 [-9999. -9999. -9999. -9999. -9999. -9999.]
 [-9999. -9999. -9999. -9999. -9999. -9999.]
 [-9999. -9999. -9999. -9999. -9999. -9999.]
 [-9999. -9999. -9999. -9999. -9999. -9999.]]
```

The second array, initially filled with `0`s, will hold the pixel value as calculated from the vectorized sliding windows (e.g., mean). Since we are not performing calculations on any edge pixels, the shape of this array is slightly smaller than that of the input raster and also dependent on the shape of the kernel. This array will be inserted into and replace the non-edge placeholder values in the output array (which we just created).

```
# Create raster array used to store window operation calculations for each pixel
(excluding boundary pixels)
aggregate = np.full((raster.shape[0] - kernel_shape[0] + 1, raster.shape[1] -
kernel_shape[1] + 1), 0)

# Set array data type
aggregate = aggregate.astype(np.float64)

# Display raster array
print(aggregate)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

Create vectorized sliding window

The next step is to generate the vectorized sliding windows. The shape of the vectorized sliding windows depends on the kernel shape, so we utilize indices and slicing to obtain the extent of a vectorized sliding window for each position in the kernel.

```
# Generate row index pairs for slicing
pairs_x = list(zip([None] + list(np.arange(1, kernel_shape[0])), list(np.arange(-
kernel_shape[0] + 1, 0)) + [None]))

# Generate column index pairs for slicing
pairs_y = list(zip([None] + list(np.arange(1, kernel_shape[1])), list(np.arange(-
kernel_shape[1] + 1, 0)) + [None]))

# Combine row and column index pairs together to get the extent for each
vectorized sliding window
combos = list(product(pairs_x, pairs_y))

# Display combined pairs
print(combos)
```

```
[((None, -2), (None, -2)), ((None, -2), (1, -1)), ((None, -2), (2, None)), ((1,
-1), (None, -2)), ((1, -1), (1, -1)), ((1, -1), (2, None)), ((2, None), (None,
-2)), ((2, None), (1, -1)), ((2, None), (2, None))]
```

Apply vectorized sliding window

Once the vectorized sliding windows are specified, we can apply them to the raster. The vectorized sliding window will get a subset of the raster array, and we can multiply all the pixel values in that subset by the corresponding value in the kernel, which is based on window's specific kernel position.

The product is then added to the array that keeps track of the running total.

```
# Create empty list to store each window operation calculation
sub_array_list = []

# Iterate through the combined pairs (which give extent of a sliding window)
for p in range(len(combos)):

    # Get the sub-array via slicing and multiply all the values by corresponding
value in kernel (based on location)
    sub_array = raster[combos[p][0][0]:combos[p][0][1], combos[p][1][0]:combos[p]
[1][1]] * kernel_array[p]

    # Add sub-array values to array storing window operation calculations
    aggregate += sub_array

    # Add sub-array to list
    sub_array_list.append(sub_array)

# View array storing window operation calculations
print(aggregate)
```

```
[[6518.4  3505.92 2670.72 4012.8 ]
 [5176.32 2163.84 1328.64 2670.72]
 [6011.52 2999.04 2163.84 3505.92]
 [9024.   6011.52 5176.32 6518.4 ]]
```

Once we get the aggregate of all windows, we can perform additional computations. In this case, we multiply all values by `2`.

```
# Get average value
aggregate = aggregate * 2

# View array storing window operation calculations
print(aggregate)
```

```
[[13036.8   7011.84  5341.44  8025.6 ]
 [10352.64  4327.68  2657.28  5341.44]
 [12023.04  5998.08  4327.68  7011.84]
 [18048.   12023.04 10352.64 13036.8 ]]
```

Window operations with predefined functions

We can also perform window operations with predefined functions, such as getting the maximum value of a pixel and its surrounding pixels. We simply take the calculations from each vectorized sliding window and get the maximum value from those calculations.

> **Tip**
>
> To get the maximum or minimum value of a pixel and its surrounding neighbors, the kernel should be filled with values of `1` so that the pixel values don't change.

```
# Get maximum value
window_maximum = np.maximum.reduce(sub_array_list)
print(window_maximum)
```

```
[[1530.66666667  954.02666667  675.62666667 1066.66666667]
 [1139.62666667  562.98666667  284.58666667  675.62666667]
 [1418.02666667  841.38666667  562.98666667  954.02666667]
 [1994.66666667 1418.02666667 1139.62666667 1530.66666667]]
```

Save output

We can insert the processed aggregate array into the output array, with each value replacing the placeholder value at its corresponding original position in the array. Recall that because edge pixels are ignored, the edge pixels of the output array will still keep their placeholder values.

In this example, each non-edge output pixel value is the average pixel value–with pixel values drawn from the input raster–of the 8 pixels surrounding that pixel and the pixel itself.

```
# Use kernel shape to determine the row and column index extent of the calculated
array
n = int((kernel_shape[0] - 1) / 2)
m = int((kernel_shape[1] - 1) / 2)

# Replace placeholder values in the output array with the corresponding values
(based on location) from the calculated array
output_rio[n:-n, m:-m] = aggregate

# Display output array
print(output_rio)
```

```
[[-9999.   -9999.   -9999.   -9999.   -9999.   -9999.  ]
 [-9999.   13036.8   7011.84  5341.44  8025.6  -9999.  ]
 [-9999.   10352.64  4327.68  2657.28  5341.44 -9999.  ]
 [-9999.   12023.04  5998.08  4327.68  7011.84 -9999.  ]
 [-9999.   18048.   12023.04 10352.64 13036.8  -9999.  ]
 [-9999.   -9999.   -9999.   -9999.   -9999.   -9999.  ]]
```

Finally, we can export the raster (click the + below to show code cell).

```
# Export raster
with rasterio.open(
        "../temp/raster_window_3x3_average.tif", "w",
        driver="GTiff",
        transform = src.transform,
        dtype=rasterio.float64,
        count=1,
        width=src.width,
        height=src.height) as dst:
    dst.write(output_rio, indexes=1)
```

Window operations with GeoWombat

We can use `GeoWombat` for window operations if we're only interested in calculating a statistic. The code to do this with `GeoWombat` is less complex and much shorter than with `rasterio`–we can simply use the `geowombat.moving()` function.

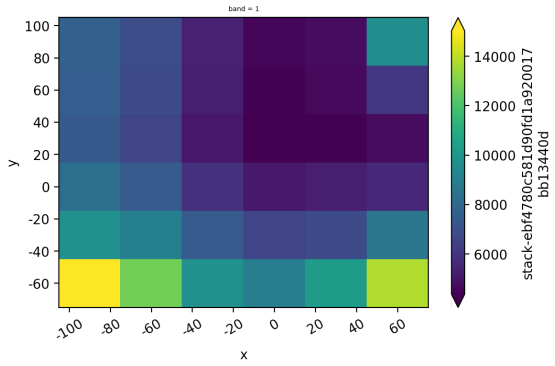
The `geowombat.moving()` function provides us with a few parameters that we can specify:

Parameter	Description
`stat`	statistic calculated (options: mean, standard deviation, variance, minimum, maximum, percentile)
`perc`	percentile used for window operation if `stat = perc`
`w`	moving window size in pixels
`nodata`	value that will be ignored in calculations

For more information this function, see the function documentation.

```
# Open file
with gw.open("../temp/window_raster.tif") as src:

    # Create plot
    fig, ax = plt.subplots(dpi = 200)

    # Calculate local average
    output_gw = src.gw.moving(stat = 'mean', w = 5, n_jobs = 4, nodata = 0)
    print(output_gw)

    # Plot raster
    output_gw.sel(band=1).gw.imshow(robust = True, ax = ax)
    plt.tight_layout(pad = 1)
```

```
<xarray.DataArray 'stack-ebf4780c581d90fd1a920017bb13440d' (band: 1, y: 6, x: 6)>
dask.array<stack, shape=(1, 6, 6), dtype=float64, chunksize=(1, 6, 6),
chunktype=numpy.ndarray>
Coordinates:
  * band     (band) int64 1
  * y        (y) float64 90.0 60.0 30.0 0.0 -30.0 -60.0
  * x        (x) float64 -90.0 -60.0 -30.0 0.0 30.0 60.0
Attributes: (12/14)
    transform:           (30.0, 0.0, -105.0, 0.0, -30.0, 105.0)
    crs:                 +init=epsg:4326
    res:                 (30.0, 30.0)
    is_tiled:            0
    nodatavals:          (nan,)
    scales:              (1.0,)
    ...                  ...
    filename:            ../temp/window_raster.tif
    resampling:          nearest
    data_are_separate:   0
    data_are_stacked:    0
    moving_stat:         mean
    moving_window_size:  5
```



Learning Objectives

- Download and utilize OpenStreetMap data

Review

- Understanding CRS codes
- Creating Points, Lines, Polygons

Accessing OSM Data in Python

What is OpenStreetMap?

OpenStreetMap (OSM) is a global collaborative (crowd-sourced) dataset and project that aims at creating a free editable map of the world containing a lot of information about our environment [1]. It contains data for example about streets, buildings, different services, and landuse to mention a few. You can view the map at www.openstreetmap.org. You can also sign up as a contributor if you want to edit the map. More details about OpenStreetMap and its contents are available in the OpenStreetMap Wiki.

OSMnx

This week we will explore a Python module called OSMnx that can be used to retrieve, construct, analyze, and visualize street networks from OpenStreetMap, and also retrieve data about Points of Interest such as restaurants, schools, and lots of different kind of services. It is also easy to conduct network routing based on walking, cycling or driving by combining OSMnx functionalities with a package called NetworkX.

To get an overview of the capabilities of the package, see an introductory video given by the lead developer of the package, Prof. Geoff Boeing: "Meet the developer: Introduction to OSMnx package by Geoff Boeing".

Download and visualize OpenStreetMap data with OSMnx

One the most useful features that OSMnx provides is an easy-to-use way of retrieving OpenStreetMap data (using OverPass API).

In this tutorial, we will learn how to download and visualize OSM data covering a specified area of interest: the neighborhood of Edgewood in Washington DC USA.

```
# Specify the name that is used to seach for the data
place_name = "Edgewood Washington, DC, USA"
```

OSM Location Boundary

Let's also plot the Polygon that represents the boundary of our area of interest (Washington DC). We can retrieve the Polygon geometry using the `ox.geocode_to_gdf` [docs](https://osmnx.readthedocs.io/en/stable/osmnx.html?highlight=geocode_to_gdf(#osmnx.geocoder.geocode_to_gdf) function.

```
# import osmnx
import osmnx as ox
import geopandas as gpd

# Get place boundary related to the place name as a geodataframe
area = ox.geocode_to_gdf(place_name)
```

As the name of the function already tells us, `gdf_from_place()`returns a GeoDataFrame based on the specified place name query.

```
# Check the data type
area
```

	geometry	bbox_north	bbox_south	bbox_east	bbox_west	place_id	osm_type	osm_id	lat	lon	display_name	class	type	importance
0	POLYGON ((-77.00892 38.92123, -77.00890 38.920...	38.934159	38.917008	-76.99358	-77.008915	282956700	relation	4634158	38.922613	-77.000537	Edgewood, Washington, District of Columbia, Un...	place	neighbourhood	0.47

Let's still verify the data type:

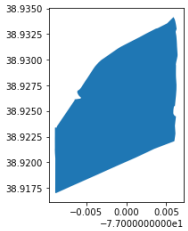
```
# Check the data type
type(area)
```

```
geopandas.geodataframe.GeoDataFrame
```

Finally, let's plot it.

```
area.plot()
```

```
<AxesSubplot:>
```



OSM Building footprints

It is also possible to retrieve other types of OSM data features with OSMnx such as buildings or points of interest (POIs). Let's download the buildings with `ox.geometries_from_place` docs function and plot them on top of our street network in Kamppi.

When fetching spesific types of geometries from OpenStreetMap using OSMnx `geometries_from_place` we also need to specify the correct tags. For getting all types of buildings, we can use the tag `building=yes`.

```
# List key-value pairs for tags
tags = {'building': True}

buildings = ox.geometries_from_place(place_name, tags)
buildings.head()
```

		addr:state	amenity	building	ele	gnis:county_id	gnis:county_name	gnis:created	gnis:edited	gnis:feature_id	gnis:import_uuid	...	shop
element_type	**osmid**												
node	**358955022**	DC	school	yes	60	001	District of Columbia	12/18/1979	01/22/2008	2062869	57871b70-0100-4405-bb30-88b2e001a944	...	NaN
	367143640	DC	NaN	yes	56	NaN	District of Columbia	NaN	NaN	2110453	57871b70-0100-4405-bb30-88b2e001a944	...	NaN
way	**52291432**	NaN	NaN	yes	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
	55321503	NaN	NaN	yes	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN
	55321504	DC	NaN	yes	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN

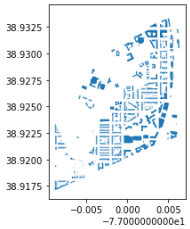
5 rows × 55 columns

We can plot the footprints quickly.

```
# Plot footprints
buildings.plot()
```

```
/home/mmann1123/anaconda3/envs/pygisbookgw/lib/python3.7/site-
packages/ipykernel/ipkernel.py:287: DeprecationWarning: `should_run_async` will
not call `transform_cell` automatically in the future. Please pass the result to
`transformed_cell` argument and any exception that happen during thetransform in
`preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
<AxesSubplot:>
```



OSM Write Features to .shp

Now let's assume we want to access this data outside of python, or have a permanent copy of our building footprints for Edgewood.

Since these objects are already `geopandas.GeoDataFrame` it's easy to save them to disk. We simply use `gpd.to_file` docs.

Important

We can't write OSM GeoDataFrames directly to disk because they contain field types (like lists) that can't be saved in .shp or .geojsons etc. Instead lets isolate only the attributes we are interested in, **including geometry** which is required.

We need to isolate just the attributes we are interested in:

```
buildings  = buildings.loc[:,buildings.columns.str.contains('addr:|geometry')]
```

```
/home/mmann1123/anaconda3/envs/pygisbookgw/lib/python3.7/site-
packages/ipykernel/ipkernel.py:287: DeprecationWarning: `should_run_async` will
not call `transform_cell` automatically in the future. Please pass the result to
`transformed_cell` argument and any exception that happen during thetransform in
`preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

Important

OSM data often contains multiple feature types like mixing points with polygons. This is a problem when we try to write it to disk.

We also need to isolate the feature type we are looking for [e.g. Multipolygon, Polygon, Point]. Since here we want building footprints we are going to keep only polygons.

```
buildings = buildings.loc[buildings.geometry.type=='Polygon']
```

Now, finally, we can write it to disk.

```
# Save footprints
buildings.to_file('../temp/edgewood_buildings.shp')
# Or save in a more open source format
#buildings.to_file('../temp/edgewood_buildings.geojson', driver='GeoJSON')
```

```
/home/mmann1123/anaconda3/envs/pygisbookgw/lib/python3.7/site-
packages/ipykernel_launcher.py:2: UserWarning: Column names longer than 10
characters will be truncated when saved to ESRI Shapefile.
```

Sources

[1] automating-gis-processes

Accessing Census and ACS Data in Python

By: Steven Chao

Learning Objectives

- Import dataframes into Python for analysis
- Perform basic dataframe column operations
- Merge dataframes using a unique key
- Group attributes based on a similar attribute
- Dissolve vector geometries based on attribute values

Review

- Data Structures
- Vector Data

Introduction

This chapter will show you how to access US Decennial Census and American Community Survey Data (ACS). We will use these basic operations in order to calculate and map poverty rates in the Commonwealth of Virginia. We will pull data from the US Census Bureau's American Community Survey (ACS) 2019 (see this page for the data).

```
# Import modules
import matplotlib.pyplot as plt
import pandas as pd
import geopandas as gpd
from census import Census
from us import states
```

Accessing Data

Import census data

Let's begin by accessing and importing census data. Importing census data into Python requires a Census API key. A key can be obtained from Census API Key. **It will provide you with a unique 40 digit text string. Please keep track of this number. Store it in a safe place.**

```
# Set API key
c = Census("CENSUS API KEY HERE")
```

```
#ignore this, I am just reading in my api key privately
with open("../../../census_api.txt", "r") as f:
    c = Census(f.read().replace('\n', ''))
```

With the Census API key set, we will access the census data at the tract level for the Commonwealth of Virginia from the 2019 ACS, specifically the `ratio of income to poverty in the past 12 months` (`C17002_001E`, total; `C17002_002E`, < 0.50; and `C17002_003E`, 0.50 - 0.99) variables and the `total population` (`B01003_001E`) variable. For more information on why these variables are used, refer to the US Census Bureau's article on how the Census Bureau measures poverty and the list of variables found in ACS.

The `census` package provides us with some easy convenience methods that allow us to obtain data for a wide variety of geographies. The FIPS code for Virginia is 51, but if needed, we can also use the `us` library to help us figure out the relevant FIPS code.

```
# Obtain Census variables from the 2019 ACS at the tract level for the
Commonwealth of Virginia (FIPS code: 51)
# C17002_001E: count of ratio of income to poverty in the past 12 months (total)
# C17002_002E: count of ratio of income to poverty in the past 12 months (< 0.50)
# C17002_003E: count of ratio of income to poverty in the past 12 months (0.50 -
0.99)
# B01003_001E: total population
# Sources: https://api.census.gov/data/2019/acs/acs5/variables.html;
https://pypi.org/project/census/
va_census = c.acs5.state_county_tract(fields = ('NAME', 'C17002_001E',
'C17002_002E', 'C17002_003E', 'B01003_001E'),
                                      state_fips = states.VA.fips,
                                      county_fips = "*",
                                      tract = "*",
                                      year = 2017)
```

Now that we have accessed the data and assigned it to a variable, we can read the data into a dataframe using the `pandas` library.

```
# Create a dataframe from the census data
va_df = pd.DataFrame(va_census)

# Show the dataframe
print(va_df.head(2))
print('Shape: ', va_df.shape)
```

```
                                      NAME  C17002_001E  C17002_002E  \
0     Census Tract 60, Norfolk city, Virginia       3947.0        284.0
1  Census Tract 65.02, Norfolk city, Virginia       3287.0        383.0

   C17002_003E  B01003_001E state county   tract
0        507.0       3947.0    51    710  006000
1        480.0       3302.0    51    710  006502
Shape:  (1907, 8)
```

By showing the dataframe, we can see that there are 1907 rows (therefore 1907 census tracts) and 8 columns.

Import Shapefile

Let's also read into Python a shapefile of the Virginia census tracts and reproject it to the UTM Zone 17N projection. (This shapefile can be downloaded on the Census Bureau's website on the Cartographic Boundary Files page or the TIGER/Line Shapefiles page.)

```
# Access shapefile of Virginia census tracts
va_tract =
gpd.read_file("https://www2.census.gov/geo/tiger/TIGER2019/TRACT/tl_2019_51_tract.
zip")

# Reproject shapefile to UTM Zone 17N
# https://spatialreference.org/ref/epsg/wgs-84-utm-zone-17n/
va_tract = va_tract.to_crs(epsg = 32617)

# Print GeoDataFrame of shapefile
print(va_tract.head(2))
print('Shape: ', va_tract.shape)

# Check shapefile projection
print("\nThe shapefile projection is: {}".format(va_tract.crs))
```

```
  STATEFP COUNTYFP TRACTCE        GEOID    NAME             NAMELSAD  MTFCC  \
0      51      700  032132  51700032132  321.32  Census Tract 321.32  G5020
1      51      700  032226  51700032226  322.26  Census Tract 322.26  G5020

  FUNCSTAT    ALAND  AWATER     INTPTLAT      INTPTLON  \
0        S  2552457       0  +37.1475176  -076.5212499
1        S  3478916  165945  +37.1625163  -076.5527816

                                      geometry
0  POLYGON ((897174.191 4119897.084, 897174.811 4...
1  POLYGON ((893470.562 4123469.385, 893542.722 4...
Shape:  (1907, 13)

The shapefile projection is: epsg:32617
```

By printing the shapefile, we can see that the shapefile also has 1907 rows (1907 tracts). This number matches with the number of census records that we have on file. Perfect!

Not so fast, though. We have a potential problem: We have the census data, and we have the shapefile of census tracts that correspond with that data, but they are stored in two separate variables (`va_df` and `va_tract` respectively)! That makes it a bit difficult to map since these two separate datasets aren't connected to each other.

Performing Dataframe Operations

Create new column from old columns to get combined FIPS code

To solve this problem, we can join the two dataframes together via a field or column that is common to both dataframes, which is referred to as a key.

Looking at the two datasets above, it appears that the `GEOID` column from `va_tract` and the `state`, `county`, and `tract` columns combined from `va_df` could serve as the unique key for joining these two dataframes together. In their current forms, this join will not be successful, as we'll need to merge the `state`, `county`, and `tract` columns from `va_df` together to make it parallel to the `GEOID` column from `va_tract`. We can simply add the columns together, much like math or the basic operators in Python, and assign the "sum" to a new column.

To create a new column–or call an existing column in a dataframe–we can use indexing with `[]` and the column name (string). (There is a different way if you want to access columns using the index number; read more about indexing and selecting data in the pandas documentation.)

```
# Combine state, county, and tract columns together to create a new string and
assign to new column
va_df["GEOID"] = va_df["state"] + va_df["county"] + va_df["tract"]
```

Printing out the first rew rows of the dataframe, we can see that the new column `GEOID` has been created with the values from the three columns combined.

```
# Print head of dataframe
va_df.head(2)
```

	NAME	C17002_001E	C17002_002E	C17002_003E	B01003_001E	state	county	tract	GEOID
0	Census Tract 60, Norfolk city, Virginia	3947.0	284.0	507.0	3947.0	51	710	006000	51710006000
1	Census Tract 65.02, Norfolk city, Virginia	3287.0	383.0	480.0	3302.0	51	710	006502	51710006502

Remove dataframe columns that are no longer needed

To reduce clutter, we can delete the `state`, `county`, and `tract` columns from `va_df` since we don't need them anymore. Remember that when we want to modify a dataframe, we must assign the modified dataframe back to the original variable (or a new one, if preferred). Otherwise, any modifications won't be saved. An alternative to assigning the dataframe back to the variable is to simply pass `inplace = True`. For more information, see the pandas help documentation on `drop`.

```
# Remove columns
va_df = va_df.drop(columns = ["state", "county", "tract"])

# Show updated dataframe
va_df.head(2)
```

	NAME	C17002_001E	C17002_002E	C17002_003E	B01003_001E	GEOID
0	Census Tract 60, Norfolk city, Virginia	3947.0	284.0	507.0	3947.0	51710006000
1	Census Tract 65.02, Norfolk city, Virginia	3287.0	383.0	480.0	3302.0	51710006502

Check column data types

The key in both dataframe must be of the same data type. Let's check the data type of the `GEOID` columns in both dataframes. If they aren't the same, we will have to change the data type of columns to make them the same.

```
# Check column data types for census data
print("Column data types for census data:\n{}".format(va_df.dtypes))

# Check column data types for census shapefile
print("\nColumn data types for census shapefile:\n{}".format(va_tract.dtypes))

# Source: https://pandas.pydata.org/pandas-
docs/stable/reference/api/pandas.DataFrame.dtypes.html
```

```
Column data types for census data:
NAME            object
C17002_001E    float64
C17002_002E    float64
C17002_003E    float64
B01003_001E    float64
GEOID           object
dtype: object

Column data types for census shapefile:
STATEFP       object
COUNTYFP      object
TRACTCE       object
GEOID         object
NAME          object
NAMELSAD      object
MTFCC         object
FUNCSTAT      object
ALAND          int64
AWATER         int64
INTPTLAT      object
INTPTLON      object
geometry    geometry
dtype: object
```

Looks like the `GEOID` columns are the same!

Merge dataframes

Now, we are ready to merge the two dataframes together, using the `GEOID` columns as the primary key. We can use the `merge` method in `GeoPandas` called on the `va_tract` shapefile dataset.

```
# Join the attributes of the dataframes together
# Source: https://geopandas.org/docs/user_guide/mergingdata.html
va_merge = va_tract.merge(va_df, on = "GEOID")

# Show result
print(va_merge.head(2))
print('Shape: ', va_merge.shape)
```

```
  STATEFP COUNTYFP TRACTCE        GEOID  NAME_x            NAMELSAD  MTFCC  \
0      51      700  032132  51700032132  321.32  Census Tract 321.32  G5020
1      51      700  032226  51700032226  322.26  Census Tract 322.26  G5020

  FUNCSTAT    ALAND  AWATER     INTPTLAT      INTPTLON  \
0        S  2552457       0  +37.1475176  -076.5212499
1        S  3478916  165945  +37.1625163  -076.5527816

                                      geometry  \
0  POLYGON ((897174.191 4119897.084, 897174.811 4...
1  POLYGON ((893470.562 4123469.385, 893542.722 4...

                                       NAME_y  C17002_001E  C17002_002E  \
0  Census Tract 321.32, Newport News city, Virginia       5025.0        161.0
1  Census Tract 322.26, Newport News city, Virginia       4167.0        736.0

   C17002_003E  B01003_001E
0        342.0       5079.0
1        559.0       4167.0
Shape:  (1907, 18)
```

Success! We still have 1907 rows, which means that all rows (or most of them) were successfully matched! Notice how the census data has been added on after the shapefile data in the dataframe.

Some additional notes about joining dataframes:

- the columns for the key do not need to have the same name.
- for this join, we had a one-to-one relationship, meaning one attribute in one dataframe matched to one (and only one) attribute in the other dataframe. Joins with a many-to-one, one-to-many, or many-to-many relationship are also possible, but in some cases, they require some special considerations. See this Esri ArcGIS help documentation on joins and relates for more information.

Subset dataframe

Now that we merged the dataframes together, we can further clean up the dataframe and remove columns that are not needed. Instead of using the `drop` method, we can simply select the columns we want to keep and create a new dataframe with those selected columns.

```
# Create new dataframe from select columns
va_poverty_tract = va_merge[["STATEFP", "COUNTYFP", "TRACTCE", "GEOID",
"geometry", "C17002_001E", "C17002_002E", "C17002_003E", "B01003_001E"]]

# Show dataframe
print(va_poverty_tract.head(2))
print('Shape: ', va_poverty_tract.shape)
```

```
  STATEFP COUNTYFP TRACTCE        GEOID  \
0      51      700  032132  51700032132
1      51      700  032226  51700032226

                                        geometry  C17002_001E  \
0  POLYGON ((897174.191 4119897.084, 897174.811 4...       5025.0
1  POLYGON ((893470.562 4123469.385, 893542.722 4...       4167.0

   C17002_002E  C17002_003E  B01003_001E
0        161.0        342.0       5079.0
1        736.0        559.0       4167.0
Shape:  (1907, 9)
```

Notice how the number of columns dropped from 13 to 9.

Dissolve geometries and get summarized statistics to get poverty statistics at the county level

Next, we will group all the census tracts within the same county (`COUNTYFP`) and aggregate the poverty and population values for those tracts within the same county. We can use the `dissolve` function in `GeoPandas`, which is the spatial version of `groupby` in `pandas`. We use `dissolve` instead of `groupby` because the former also groups and merges all the geometries (in this case, census tracts) within a given group (in this case, counties).

```
# Dissolve and group the census tracts within each county and aggregate all the
values together
# Source: https://geopandas.org/docs/user_guide/aggregation_with_dissolve.html
va_poverty_county = va_poverty_tract.dissolve(by = 'COUNTYFP', aggfunc = 'sum')

# Show dataframe
print(va_poverty_county.head(2))
print('Shape: ', va_poverty_county.shape)
```

```
                                             geometry  C17002_001E  \
COUNTYFP
001       POLYGON ((971901.668 4160101.088, 971814.409 4...      32345.0
003       POLYGON ((734957.267 4207640.156, 734931.249 4...      97587.0

           C17002_002E  C17002_003E  B01003_001E
COUNTYFP
001             2423.0       3993.0      32840.0
003             5276.0       4305.0     105105.0
Shape:  (133, 5)
```

Notice that we got the number of rows down from 1907 to 133.

Perform column math to get poverty rates

We can estimate the poverty rate by dividing the sum of `C17002_002E` (ratio of income to poverty in the past 12 months, < 0.50) and `C17002_003E` (ratio of income to poverty in the past 12 months, 0.50 - 0.99) by `B01003_001E` (total population).

Side note: Notice that `C17002_001E` (ratio of income to poverty in the past 12 months, total), which theoretically should count everyone, does not exactly match up with `B01003_001E` (total population). We'll disregard this for now since the difference is not too significant.

```
# Get poverty rate and store values in new column
va_poverty_county["Poverty_Rate"] = (va_poverty_county["C17002_002E"] +
va_poverty_county["C17002_003E"]) / va_poverty_county["B01003_001E"] * 100

# Show dataframe
va_poverty_county.head(2)
```

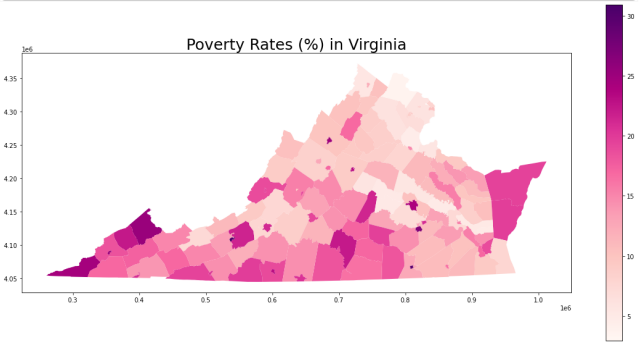
COUNTYFP	geometry	C17002_001E	C17002_002E	C17002_003E	B01003_001E	Poverty_Rate
001	POLYGON ((971901.668 4160101.088, 971814.409 4...	32345.0	2423.0	3993.0	32840.0	19.537150
003	POLYGON ((734957.267 4207640.156, 734931.249 4...	97587.0	5276.0	4305.0	105105.0	9.115646

Plotting Results

Finally, since we have the spatial component connected to our census data, we can plot the results!

```
# Create subplots
fig, ax = plt.subplots(1, 1, figsize = (20, 10))

# Plot data
# Source: https://geopandas.readthedocs.io/en/latest/docs/user_guide/mapping.html
va_poverty_county.plot(column = "Poverty_Rate",
                       ax = ax,
                       cmap = "RdPu",
                       legend = True)

# Stylize plots
plt.style.use('bmh')

# Set title
ax.set_title('Poverty Rates (%) in Virginia', fontdict = {'fontsize': '25',
'fontweight' : '3'})
```

```
Text(0.5, 1.0, 'Poverty Rates (%) in Virginia')
```



Learning Objectives

- How to open multiple common remotely sensed image types
- Handle RGB, BGR, LandSat, PlanetScope images and other sensor types
- Mosaic multiple remotely sensed images
- Create a time series stack
- Write files to disk

Review

- Data Structures
- Raster Data

Reading/Writing Remote Sensed Images

GeoWombat's file opening is meant to mimic Xarray and Rasterio. That is, rasters are typically opened with a context manager using the function `geowombat.open`. GeoWombat uses `xarray.open_rasterio` to load data into an `xarray.DataArray`. In GeoWombat, the data are always chunked, meaning the data are always loaded as Dask arrays. As with `xarray.open_rasterio`, the opened DataArrays always have at least 1 band.

Opening a single image

Opening an image with default settings looks similar to `xarray.open_rasterio` and `rasterio.open`. `geowombat.open` expects a file name (`str` or `pathlib.Path`).

```
import geowombat as gw
from geowombat.data import l8_224078_20200518
import matplotlib.pyplot as plt

with gw.open(l8_224078_20200518) as src:
    print(src)
```

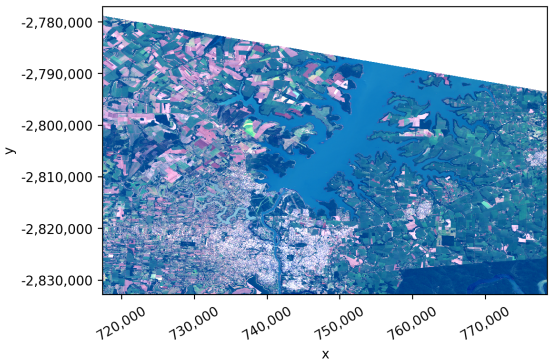
```
<xarray.DataArray (band: 3, y: 1860, x: 2041)>
dask.array<open_rasterio-cd30c5f7c3868320497352519d525008<this-array>, shape=(3,
1860, 2041), dtype=uint16, chunksize=(1, 256, 256), chunktype=numpy.ndarray>
Coordinates:
  * band     (band) int64 1 2 3
  * y        (y) float64 -2.777e+06 -2.777e+06 ... -2.833e+06 -2.833e+06
  * x        (x) float64 7.174e+05 7.174e+05 7.174e+05 ... 7.785e+05 7.786e+05
Attributes:
    transform:          (30.0, 0.0, 717345.0, 0.0, -30.0, -2776995.0)
    crs:                +init=epsg:32621
    res:                (30.0, 30.0)
    is_tiled:           1
    nodatavals:         (nan, nan, nan)
    scales:             (1.0, 1.0, 1.0)
    offsets:            (0.0, 0.0, 0.0)
    AREA_OR_POINT:      Area
    filename:           /home/mmann1123/anaconda3/envs/pygisbookgw/lib/python...
    resampling:         nearest
    data_are_separate:  0
    data_are_stacked:   0
```

In the example above, `src` is an `xarray.DataArray`. Thus, printing the object will display the underlying Dask array dimensions and chunks, the DataArray named coordinates, and the DataArray attributes.

It automatically converts the coordinates stored in `x` and `y`, and the different bands are stored in `band`. To select a single band we can simply select it with `src.sel(band=1)`.

Let's plot out what we have while removing missing values, stored at `0`, and switch the band order around to be RGB.

```
fig, ax = plt.subplots(dpi=200)
with gw.open(l8_224078_20200518) as src:
    src.where(src != 0).sel(band=[3, 2, 1]).gw.imshow(robust=True, ax=ax)
plt.tight_layout(pad=1)
```



Opening multiple bands as a stack

Often, satellite bands will be stored in separate raster files. To open the files as one DataArray, specify a list instead of a file name.

```
from geowombat.data import l8_224078_20200518_B2, l8_224078_20200518_B3,
l8_224078_20200518_B4

with gw.open([l8_224078_20200518_B2, l8_224078_20200518_B3,
l8_224078_20200518_B4]) as src:
    print(src)
```

```
<xarray.DataArray (time: 3, band: 1, y: 1860, x: 2041)>
dask.array<concatenate, shape=(3, 1, 1860, 2041), dtype=uint16, chunksize=(1, 1,
256, 256), chunktype=numpy.ndarray>
Coordinates:
  * band     (band) int64 1
  * y        (y) float64 -2.777e+06 -2.777e+06 ... -2.833e+06 -2.833e+06
  * x        (x) float64 7.174e+05 7.174e+05 7.174e+05 ... 7.785e+05 7.786e+05
  * time     (time) int64 1 2 3
Attributes:
    transform:          (30.0, 0.0, 717345.0, 0.0, -30.0, -2776995.0)
    crs:                +init=epsg:32621
    res:                (30.0, 30.0)
    is_tiled:           1
    nodatavals:         (nan,)
    scales:             (1.0,)
    offsets:            (0.0,)
    AREA_OR_POINT:      Point
    filename:           ['LC08_L1TP_224078_20200518_20200518_01_RT_B2.TIF', '...
    data_are_separate:  1
    data_are_stacked:   1
```

By default, GeoWombat will stack multiple files by time. So, to stack multiple bands with the same timestamp, change the `stack_dim` keyword.

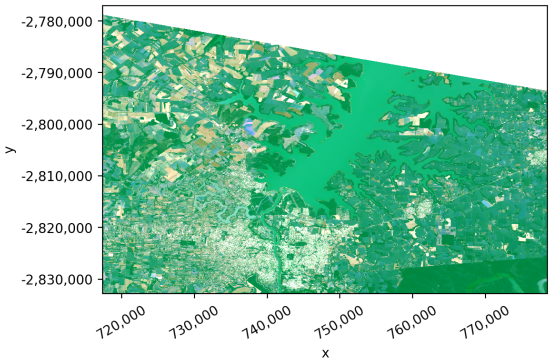
Also note the use of `band_names` parameter. Here we can set it to anything we want for instance `['blue','green','red']`.

```
from geowombat.data import l8_224078_20200518_B2, l8_224078_20200518_B3,
l8_224078_20200518_B4

with gw.open(
    [l8_224078_20200518_B2, l8_224078_20200518_B3, l8_224078_20200518_B4],
    stack_dim="band",
    band_names=[1, 2, 3],
) as src:
    print(src)
```

```
<xarray.DataArray (band: 3, y: 1860, x: 2041)>
dask.array<concatenate, shape=(3, 1860, 2041), dtype=uint16, chunksize=(1, 256,
256), chunktype=numpy.ndarray>
Coordinates:
  * band     (band) int64 1 2 3
  * y        (y) float64 -2.777e+06 -2.777e+06 ... -2.833e+06 -2.833e+06
  * x        (x) float64 7.174e+05 7.174e+05 7.174e+05 ... 7.785e+05 7.786e+05
Attributes:
    transform:          (30.0, 0.0, 717345.0, 0.0, -30.0, -2776995.0)
    crs:                +init=epsg:32621
    res:                (30.0, 30.0)
    is_tiled:           1
    nodatavals:         (nan,)
    scales:             (1.0,)
    offsets:            (0.0,)
    AREA_OR_POINT:      Point
    filename:           ['LC08_L1TP_224078_20200518_20200518_01_RT_B2.TIF', '...
    data_are_separate:  1
    data_are_stacked:   1
```

You will see this looks the same as the multiband raster:

```
fig, ax = plt.subplots(dpi=200)
with gw.open(
    [l8_224078_20200518_B2, l8_224078_20200518_B3, l8_224078_20200518_B4],
    stack_dim="band",
    band_names=['blue','green','red'],
) as src:
    src.where(src != 0).sel(band=['red','blue','green']).gw.imshow(robust=True,
ax=ax)
plt.tight_layout(pad=1)
```



> **Note**
>
> If time names are not specified with `stack_dim` = 'time', GeoWombat will attempt to parse dates from the file names. This could incur significant overhead when the file list is long. Therefore, it is good practice to specify the time names.

Overhead required to parse file names

```
with gw.open(long_file_list, stack_dim='time') as src:
    ...
```

No file parsing overhead

```
with gw.open(long_file_list, time_names=my_time_names, stack_dim='time') as src:
    ...
```

Opening images from different sensors

One of many complications of using remotely sensed data is that there are so many different sensors such as LandSat, Sentinel, PlantScope etc each with their own band order and properties. Geowombat makes this much easier by providing a broad list of potential sensor configurations. Read in more detail about sensor configurations here. For this section, let's keep things simple and show you how to open a Sentinel 2 image using the configuration manager, frankly, it's pretty easy:

```
with gw.config.update(sensor='s2'):
    with gw.open('filepath.tif') as src:
        print(src.band)
```

To see all available sensor names, use the **avail_sensors** property.

```
with gw.open('filepath.tif') as src:
    for sensor_name in src.gw.avail_sensors:
        print(sensor_name)
```

Opening multiple bands as a mosaic

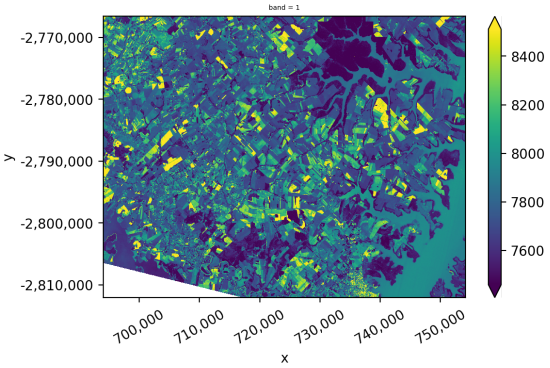
When a list of files are given, GeoWombat will stack the data by default. To mosaic multiple files into the same band coordinate, use the **mosaic** keyword.

```
from geowombat.data import l8_224077_20200518_B2, l8_224078_20200518_B2

with gw.open([l8_224077_20200518_B2, l8_224078_20200518_B2],
             mosaic=True) as src:
    print(src)
```

```
<xarray.DataArray (y: 1515, x: 2006, band: 1)>
dask.array<where, shape=(1515, 2006, 1), dtype=uint16, chunksize=(256, 256, 1),
chunktype=numpy.ndarray>
Coordinates:
  * y        (y) float64 -2.767e+06 -2.767e+06 ... -2.812e+06 -2.812e+06
  * x        (x) float64 6.94e+05 6.940e+05 6.941e+05 ... 7.541e+05 7.542e+05
  * band     (band) int64 1
Attributes:
    transform:          (30.0, 0.0, 694005.0, 0.0, -30.0, -2766615.0)
    crs:                +init=epsg:32621
    res:                (30.0, 30.0)
    is_tiled:           1
    nodatavals:         (nan,)
    scales:             (1.0,)
    offsets:            (0.0,)
    AREA_OR_POINT:      Point
    filename:           ['LC08_L1TP_224077_20200518_20200518_01_RT_B2.TIF', '...
    resampling:         nearest
    data_are_separate:  1
    data_are_stacked:   0
```

Now let's take a look at the mosaiced band 2 image values.

```
fig, ax = plt.subplots(dpi=200)
with gw.open([l8_224077_20200518_B2, l8_224078_20200518_B2],
             mosaic=True) as src:
    src.where(src != 0).sel(band=1).gw.imshow(robust=True, ax=ax)
plt.tight_layout(pad=1)
```



Create a Time Series Stack

Let's pretend for a moment that we have a time series of images from the same tile. We can stack them by passing a list of file names `[l8_224078_20200518, l8_224078_20200518]`, it also helps to be specific and assign `time_names=['t1', 't2']`, and specify which dimension we want to stack our data along with `stack_dim='time'`.

```
with gw.open([l8_224078_20200518, l8_224078_20200518],
             band_names=['blue', 'green', 'red'],
             time_names=['t1', 't2'],
             stack_dim='time') as src:
    print(src)
```

```
<xarray.DataArray (time: 2, band: 3, y: 1860, x: 2041)>
dask.array<concatenate, shape=(2, 3, 1860, 2041), dtype=uint16, chunksize=(1, 1,
256, 256), chunktype=numpy.ndarray>
Coordinates:
  * band     (band) <U5 'blue' 'green' 'red'
  * y        (y) float64 -2.777e+06 -2.777e+06 ... -2.833e+06 -2.833e+06
  * x        (x) float64 7.174e+05 7.174e+05 7.174e+05 ... 7.785e+05 7.786e+05
  * time     (time) <U2 't1' 't2'
Attributes:
    transform:          (30.0, 0.0, 717345.0, 0.0, -30.0, -2776995.0)
    crs:                +init=epsg:32621
    res:                (30.0, 30.0)
    is_tiled:           1
    nodatavals:         (nan, nan, nan)
    scales:             (1.0, 1.0, 1.0)
    offsets:            (0.0, 0.0, 0.0)
    AREA_OR_POINT:      Area
    filename:           ['LC08_L1TP_224078_20200518_20200518_01_RT.TIF', 'LC0...
    resampling:         nearest
    data_are_separate:  1
    data_are_stacked:   1
```

Writing DataArrays to file

GeoWombat's I/O can be accessed through the `to_vrt` and `to_raster` functions. These functions use Rasterio's `write` and Dask.array `store` functions as I/O backends. In the examples below, `src` is an `xarray.DataArray` with the necessary transform information to write to an image file.

Write to a VRT file.

```
from geowombat.data import l8_224077_20200518_B4

# Transform the data to lat/lon
with gw.config.update(ref_crs=4326):

    with gw.open(l8_224077_20200518_B4) as src:

        # Write the data to a VRT
        gw.to_vrt(src, 'lat_lon_file.vrt')
```

Write to a raster file.

```
import geowombat as gw

with gw.open(l8_224077_20200518_B4) as src:

    # Xarray drops attributes
    attrs = src.attrs.copy()

    # Apply operations on the DataArray
    src = src * 10.0

    src.attrs = attrs

    # Write the data to a GeoTiff
    src.gw.to_raster('output.tif',
                     verbose=1,
                     n_workers=4,    # number of process workers sent to ``concurrent.futures``
                     n_threads=2,    # number of thread workers sent to ``dask.compute``
                     n_chunks=200)   # number of window chunks to send as concurrent futures
```

Learning Objectives

- Reproject remotely sensed data (change CRS)
- Reproject on-the-fly
- Understand resampling options

Review

- What is a CRS
- Understanding CRS codes
- Raster CRS

Configuration manager

What is a context manager?

In short, a context manager ensures proper file closing using with statements. But it also allows us to set up default behavoirs for opening and writing our images.

What is the purpose of GeoWombat's context manager?

The examples shown in reading remotely sensed data opened the entire raster as DataArrays as they were stored on file. The configuration manager allows easy control over opened raster dimensions, alignment, and transformations.

For instance you might want to set the bound (extent) of your analysis. By setting bounds with the configuration manager we will minimize our overhead (less data processed) and uniformly treat all images we process.

How do I use it?

To use GeoWombat's configuration manager, just call `geowombat.config.update` before opening a file. For example,

```
import geowombat as gw

with gw.config.update(<keywords>...):

# Every file opened within the configuration block will use
# configuration keywords
with gw.open('image.tif') as src:
    # do something
```

`geowombat.config.update` stores keywords in a dictionary. To see all GeoWombat configuration keywords, just iterate over the dictionary.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518

# Using the manager without keywords will set defaults
with gw.config.update():
    with gw.open(l8_224078_20200518) as src:
        for k, v in src.gw.config.items():
            print('Keyword:', k.ljust(15), 'Value:', v)
```

```
Keyword: with_config     Value: True
Keyword: ignore_warnings Value: False
Keyword: sensor          Value: None
Keyword: scale_factor    Value: 1.0
Keyword: nodata          Value: None
Keyword: ref_image       Value: None
Keyword: ref_bounds      Value: None
Keyword: ref_crs         Value: None
Keyword: ref_res         Value: None
Keyword: ref_tar         Value: None
Keyword: blockxsize      Value: 512
Keyword: blockysize      Value: 512
Keyword: compress        Value: None
Keyword: driver          Value: GTiff
Keyword: tiled           Value: True
Keyword: bigtiff         Value: NO
Keyword: l57_angles_path Value: None
Keyword: l8_angles_path  Value: None
```

Reference settings: CRS

Configuration keywords beginning with **ref** are the most important commands when opening rasters. For example, to transform the CRS of the data on-the-fly, use **ref_crs**. For more on Coordinate Reference Systems, see here.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518

proj4 = "+proj=aea +lat_1=-5 +lat_2=-42 +lat_0=-32 +lon_0=-60 +x_0=0 +y_0=0
+ellps=aust_SA +units=m +no_defs "
0
# Without the manager
with gw.open(l8_224078_20200518) as src:
    print(src.crs)

# With the manager
with gw.config.update(ref_crs=proj4):
    with gw.open(l8_224078_20200518) as src:
        print(src.crs)
```

```
+init=epsg:32621
+proj=aea +lat_0=-32 +lon_0=-60 +lat_1=-5 +lat_2=-42 +x_0=0 +y_0=0 +ellps=aust_SA
+units=m +no_defs=True
```

Reference settings: Cell size

It is possible to combine multiple configuration keywords. In the example below, the raster CRS is transformed from UTM to Albers Equal Area with a resampled cell size of 100m x 100m.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518

# Without the manager
with gw.open(l8_224078_20200518) as src:
    print(src.gw.celly, src.gw.cellx)

# With the manager
with gw.config.update(ref_crs=proj4, ref_res=(100, 100)):
    with gw.open(l8_224078_20200518) as src:
        print(src.gw.celly, src.gw.cellx)
```

```
30.0 30.0
100.0 100.0
```

Reference settings: Bounds

To subset an image, specify bounds as a **tuple** of (left, bottom, right, top) or a rasterio **BoundingBox** object.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518
from rasterio.coords import BoundingBox

bounds = BoundingBox(left=724634.17, bottom=-2806501.39, right=737655.48,
top=-2796221.42)

# or
# bounds = (724634.17, -2806501.39, 737655.48, -2796221.42)

# Without the manager
with gw.open(l8_224078_20200518) as src:
    print(src.gw.bounds)

# With the manager
with gw.config.update(ref_bounds=bounds):
    with gw.open(l8_224078_20200518) as src:
        print(src.gw.bounds)
```

```
(717345.0, -2832795.0, 778575.0, -2776995.0)
```

```
(724634.17, -2806481.42, 737654.17, -2796221.42)
```

Reference settings: Snap Raster Target

By default, the bounding subset will be returned by the upper left coordinates of the bounds, potentially shifting cell alignment with the reference raster. To subset a raster and align it to the same grid, use the **ref_tar** keyword. This is equivalent to a “snap raster” in ArcGIS.

```
with gw.config.update(ref_bounds=bounds, ref_tar=rgbn):

    with gw.open(rgbn) as src:
        print(src)
```

Reference Image

To use another image as a reference, just set **ref_image**. Then, the opened file’s bounds, CRS, and cell size will be transformed to match those of the reference image.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518, l8_224077_20200518_B2

# Without the manager
with gw.open(l8_224078_20200518) as src:
    print(src.gw.bounds)

with gw.open(l8_224077_20200518_B2) as src:
    print(src.gw.bounds)

# With the manager
with gw.config.update(ref_image=l8_224077_20200518_B2):
    with gw.open(l8_224078_20200518) as src:
        print(src.gw.bounds)
```

```
(717345.0, -2832795.0, 778575.0, -2776995.0)
```

```
(694005.0, -2812065.0, 754185.0, -2766615.0)
(694005.0, -2812065.0, 754185.0, -2766615.0)
```

Reference settings: Sensors

Because rasters are opened as DataArrays, the band coordinates will be named. By default, the bands will be named by their index position (starting at 1). It might, however, be more intuitive to store the band names as strings, where the names correspond to the sensor wavelengths. In GeoWombat, you can set the band names explicitly upon opening a file by using the :func:`geowombat.open` **band_names** keyword. Alternatively, if the sensor is known (and supported by GeoWombat), then you can set the band names by specifying the sensor name in the configuration settings.

> **Note**
>
> In the example below, the example raster comes from a Landsat image. However, only the visible (blue, green, and red) wavelengths are stored. Thus, we use ‘rgb’ as the sensor name. If we had a full 6-band Landsat 7 image, for example, we could use the ‘l7’ sensor flag.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518

# Without the manager
with gw.open(l8_224078_20200518) as src:
    print(src.band)

# With the manager
with gw.config.update(sensor='bgr'):
    with gw.open(l8_224078_20200518) as src:
        print(src.band)
```

```
<xarray.DataArray 'band' (band: 3)>
array([1, 2, 3])
Coordinates:
  * band     (band) int64 1 2 3
<xarray.DataArray 'band' (band: 3)>
array(['blue', 'green', 'red'], dtype='<U5')
Coordinates:
  * band     (band) <U5 'blue' 'green' 'red'
```

To see all available sensor names, use the **avail_sensors** property.

```
with gw.open(l8_224078_20200518) as src:
    for sensor_name in src.gw.avail_sensors:
        print(sensor_name)
```

For a short description of the sensor, use the **sensor_names** property.

```
with gw.open(l8_224078_20200518) as src:
    for sensor_name, description in src.gw.sensor_names.items():
        print('{}: {}'.format(sensor_name.ljust(15), description))
```

```
rgb             : red, green, and blue
rgbn            : red, green, blue, and NIR
bgr             : blue, green, and red
bgrn            : blue, green, red, and NIR
l5              : Landsat 5 Thematic Mapper (TM)
l7              : Landsat 7 Enhanced Thematic Mapper Plus (ETM+) without
panchromatic and thermal bands
l7th            : Landsat 7 Enhanced Thematic Mapper Plus (ETM+) with thermal band
l7mspan         : Landsat 7 Enhanced Thematic Mapper Plus (ETM+) with panchromatic
band
l7pan           : Landsat 7 panchromatic band
l8              : Landsat 8 Operational Land Imager (OLI) and Thermal Infrared
Sensor (TIRS) without panchromatic and thermal bands
l8l7            : Landsat 8 Operational Land Imager (OLI) and Thermal Infrared
Sensor (TIRS) with 6 Landsat 7-like bands
l8l7mspan       : Landsat 8 Operational Land Imager (OLI) and panchromatic band
with 6 Landsat 7-like bands
l8th            : Landsat 8 Operational Land Imager (OLI) and Thermal Infrared
Sensor (TIRS) with thermal band
l8pan           : Landsat 8 panchromatic band
s2              : Sentinel 2 Multi-Spectral Instrument (MSI) without 3 60m bands
(coastal, water vapor, cirrus)
s2f             : Sentinel 2 Multi-Spectral Instrument (MSI) with 3 60m bands
(coastal, water vapor, cirrus)
s2l7            : Sentinel 2 Multi-Spectral Instrument (MSI) with 6 Landsat 7-like
bands
s210            : Sentinel 2 Multi-Spectral Instrument (MSI) with 4 10m (visible +
NIR) bands
s220            : Sentinel 2 Multi-Spectral Instrument (MSI) with 6 20m bands
s2cloudless     : Sentinel 2 Multi-Spectral Instrument (MSI) with 10 bands for
s2cloudless
ps              : PlanetScope with 4 (visible + NIR) bands
qb              : Quickbird with 4 (visible + NIR) bands
ik              : IKONOS with 4 (visible + NIR) bands
```

The following is a list of all available sensor names. This documentation may become out of date, if so please refer to geowombat/core/properties.py for the full list.

Abbreviated Name	Description
'rgb'	red, green, and blue
'rgbn'	red, green, blue, and NIR
'bgr'	blue, green, and red
'bgrn'	blue, green, red, and NIR
'l5'	Landsat 5 Thematic Mapper (TM)
'l7'	Landsat 7 Enhanced Thematic Mapper Plus (ETM+) without panchromatic and thermal bands
'l7th'	Landsat 7 Enhanced Thematic Mapper Plus (ETM+) with thermal band
'l7mspan'	Landsat 7 Enhanced Thematic Mapper Plus (ETM+) with panchromatic band
'l7pan'	Landsat 7 panchromatic band
'l8'	Landsat 8 Operational Land Imager (OLI) and Thermal Infrared Sensor (TIRS) without panchromatic and thermal bands
'l8l7'	Landsat 8 Operational Land Imager (OLI) and Thermal Infrared Sensor (TIRS) with 6 Landsat 7-like bands
'l8l7mspan'	Landsat 8 Operational Land Imager (OLI) and panchromatic band with 6 Landsat 7-like bands
'l8th'	Landsat 8 Operational Land Imager (OLI) and Thermal Infrared Sensor (TIRS) with thermal band
'l8pan'	Landsat 8 panchromatic band
's2'	Sentinel 2 Multi-Spectral Instrument (MSI) without 3 60m bands (coastal, water vapor, cirrus)
's2f'	Sentinel 2 Multi-Spectral Instrument (MSI) with 3 60m bands (coastal, water vapor, cirrus)
's2l7'	Sentinel 2 Multi-Spectral Instrument (MSI) with 6 Landsat 7-like bands
's210'	Sentinel 2 Multi-Spectral Instrument (MSI) with 4 10m (visible + NIR) bands
's220'	Sentinel 2 Multi-Spectral Instrument (MSI) with 6 20m bands
's2cloudless'	Sentinel 2 Multi-Spectral Instrument (MSI) with 10 bands for s2cloudless
'ps'	PlanetScope with 4 (visible + NIR) bands
'qb'	Quickbird with 4 (visible + NIR) bands
'ik'	IKONOS with 4 (visible + NIR) bands

Learning Objectives

- Handle missing values
- Setting missing values
- Replacing values

Review

- Opening remotely sensed data

Editing Rasters and Remotely Sensed Data

Setting 'no data' Values

The `xarray.DataArray.where` function masks data by setting nans, as demonstrated by the example below.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518

# Zeros are replaced with nans
with gw.open(l8_224078_20200518) as src:
    data = src.where(src != 0)
```

Setting 'no data' Values with Scaling

In GeoWombat, we use `xarray.where` and `xarray.DataArray.where` along with optional scaling in the `set_nodata` function. In this example, we set zeros as 65535 and scale all other values from a [0,10000] range to [0,1].

```
import geowombat as gw
from geowombat.data import l8_224078_20200518

# Set the 'no data' value and scale all other values
with gw.open(l8_224078_20200518) as src:
    data = src.gw.set_nodata(0, 65535, (0, 1), 'float64', scale_factor=0.0001)
```

Replace values

The GeoWombat `replace` function mimics `pandas.DataFrame.replace`.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518

# Replace 1 with 10
with gw.open(l8_224078_20200518) as src:
    data = src.gw.replace({1: 10})
```

Note

The `replace` function is typically used with categorical data.

Learning Objectives

- Visualize RGB images from remotely sensed data
- Visualize true-color and false-color composites
-

Review

- Data Structures
- Raster Data
- Opening Remotely Sensed Data

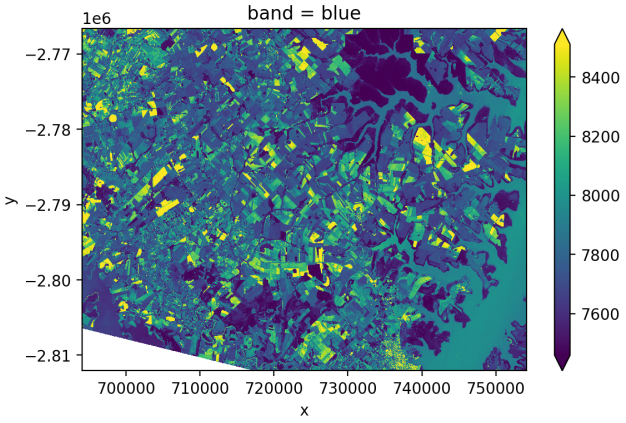
Plot Remote Sensed Images

Import required modules and data.

```
# Import GeoWombat
import geowombat as gw

# import plotting
from pathlib import Path
import matplotlib.pyplot as plt
import matplotlib.patheffects as pe
```

Plot a Single Band Image

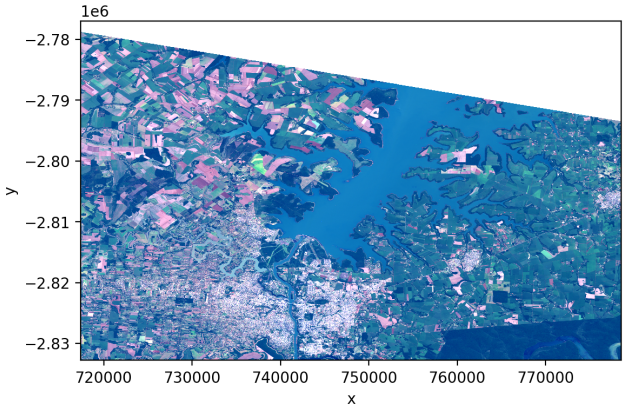
```
from geowombat.data import l8_224077_20200518_B2

fig, ax = plt.subplots(dpi=200)

with gw.open(l8_224077_20200518_B2,
             band_names=['blue']) as src:
    src.where(src != 0).sel(band='blue').plot.imshow(robust=True, ax=ax)
plt.tight_layout(pad=1)
```



Plot a True Color LandSat Image

Here we open the image, missing data is removed with `.where(src != 0)`, remember the bands in this file are stored in reverse order (blue, green, red), so we put them back into order `.sel(band=[3, 2, 1])`.

```
# load example data
from geowombat.data import l8_224078_20200518

fig, ax = plt.subplots(dpi=200)
with gw.open(l8_224078_20200518) as src:
    src.where(src != 0).sel(band=[3, 2, 1]).plot.imshow(robust=True, ax=ax)
plt.tight_layout(pad=1)
```



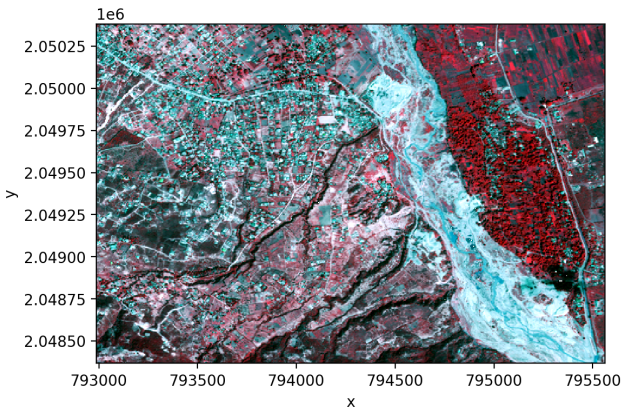
Plot False Color Composites

We can use the red, green, and blue channels to show different parts of the spectrum. This allows us for instance to “see” near-infrared (nir). Moreover certain combinations of bands allow us to better identify vegetation, urban environments, water, etc. There are many false colored composites that can be used to highlight different features.

Color Infrared (vegetation)

Here we will look at a common false color combo to assigns the nir band to the color red. This make vegetation appear bright red.

```
from geowombat.data import rgbn

fig, ax = plt.subplots(dpi=200)

with gw.open(rgbn,
             band_names=['red','green','blue','nir'],) as src:
    src.where(src != 0).sel(band=['nir','red', 'green']).plot.imshow(robust=True,
ax=ax)
plt.tight_layout(pad=1)
plt.savefig("rgb_plot.png", dpi=150)
```



Common Band Combinations for Landsat 8

Name	Band Combination
Natural Color	4 3 2
False Color (urban)	7 6 4
Color Infrared (vegetation)	5 4 3
Agriculture	6 5 2
Atmospheric Penetration	7 6 5
Healthy Vegetation	5 6 2
Land/Water	5 6 4
Natural With Atmospheric Removal	7 5 3
Shortwave Infrared	7 5 4
Vegetation Analysis	6 5 4

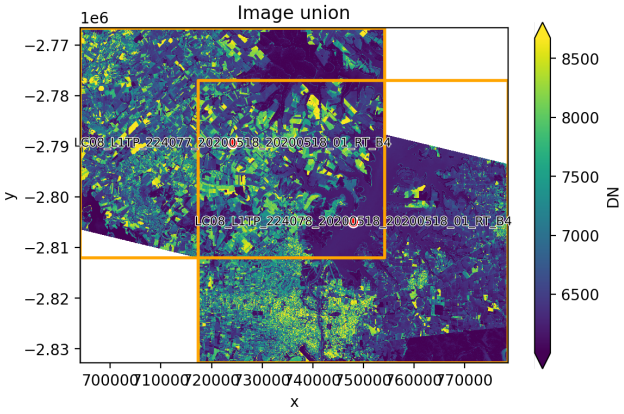
Plot LandSat Tile Footprints

Here we set up a more complicated plotting function for near IR 'nir'. Note the use of `footprint_grid`.

```
from geowombat.data import l8_224077_20200518_B4, l8_224078_20200518_B4

def plot(bounds_by, ref_image=None, cmap='viridis'):
    fig, ax = plt.subplots(dpi=200)
    with gw.config.update(ref_image=ref_image):
        with gw.open([l8_224077_20200518_B4, l8_224078_20200518_B4],
                     band_names=['nir'],
                     chunks=256,
                     mosaic=True,
                     bounds_by=bounds_by) as srca:
            # Plot the NIR band
            srca.where(srca != 0).sel(band='nir').plot.imshow(robust=True,
cbar_kwargs={'label': 'DN'}, ax=ax)
            # Plot the image chunks
            srca.gw.chunk_grid.plot(color='none', edgecolor='k', ls='-', lw=0.5,
ax=ax)
            # Plot the image footprints
            srca.gw.footprint_grid.plot(color='none', edgecolor='orange', lw=2,
ax=ax)
            # Label the image footprints
            for row in srca.gw.footprint_grid.itertuples(index=False):
                ax.scatter(row.geometry.centroid.x, row.geometry.centroid.y,
                           s=50, color='red', edgecolor='white', lw=1)
                ax.annotate(row.footprint.replace('.TIF', ''),
                            (row.geometry.centroid.x, row.geometry.centroid.y),
                            color='black',
                            size=8,
                            ha='center',
                            va='center',
                            path_effects=[pe.withStroke(linewidth=1,
foreground='white')])
            # Set the display bounds
            ax.set_ylim(srca.gw.footprint_grid.total_bounds[1]-10,
srca.gw.footprint_grid.total_bounds[3]+10)
            ax.set_xlim(srca.gw.footprint_grid.total_bounds[0]-10,
srca.gw.footprint_grid.total_bounds[2]+10)
    title = f'Image {bounds_by}' if bounds_by else
str(Path(ref_image).name.split('.')[0]) + ' as reference'
    size = 12 if bounds_by else 8
    ax.set_title(title, size=size)
    plt.tight_layout(pad=1)
```

The two plots below illustrate how two images can be mosaicked. The orange grids highlight the image footprints while the black grids illustrate the `DataArray` chunks.

```
plot('union')
```



Learning Objectives

- Reproject remotely sensed data (change CRS)
- Reproject on-the-fly
- Understand resampling options

Review

- What is a CRS
- Understanding CRS codes
- Raster CRS

Remote Sensing Coordinate Reference Systems

Image projections can be transformed in GeoWombat using the configuration manager (see Config Manager). With the configuration manager, the CRS is transformed using rasterio CRS and virtual warping. For references, see Spatial Reference and epsg.io.

View Image Coordinate Reference System & Properties

In the following we will print out the properties relevant to CRS for the red, green blue image. The CRS can be accessed from the xarray.DataArray attributes.

```
import geowombat as gw
from geowombat.data import rgbn

with gw.open(rgbn) as src:
    print(src.transform)
    print(src.gw.transform)
    print(src.crs)
    print(src.resampling)
    print(src.res)
    print(src.gw.cellx, src.gw.celly)
```

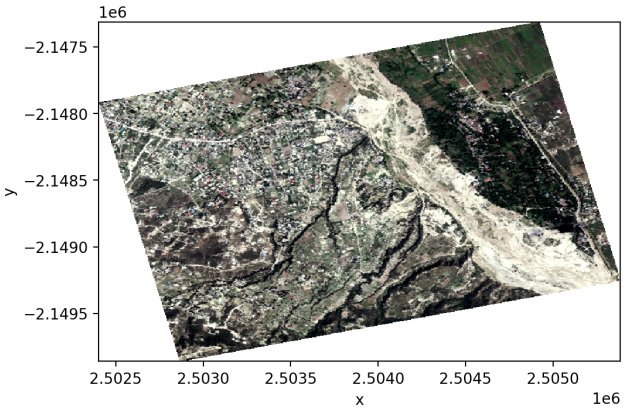
```
(5.0, 0.0, 792988.0, 0.0, -5.0, 2050382.0)
(5.0, 0.0, 792988.0, 0.0, -5.0, 2050382.0)
+init=epsg:32618
nearest
(5.0, 5.0)
5.0 5.0
```

Transforming a CRS On-The-Fly

To transform the CRS, use the context manager. In this example, an proj4 code is used. See understanding CRS codes for more details.

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(dpi=200)

proj4 = "+proj=aea +lat_1=20 +lat_2=60 +lat_0=40 +lon_0=-96 +x_0=0 +y_0=0
+datum=NAD83 +units=m +no_defs"

with gw.config.update(ref_crs=proj4):
    with gw.open(rgbn) as src:
        print(src.transform)
        print(src.crs)
        print(src.resampling)
        print(src.res)
        src.where(src != 0).sel(band=[3,2,1]).plot.imshow(robust=True, ax=ax)

plt.tight_layout(pad=1)
```

```
(5.0, 0.0, 2502400.7632678417, 0.0, -5.0, -2147313.7330151177)
+proj=aea +lat_0=40 +lon_0=-96 +lat_1=20 +lat_2=60 +x_0=0 +y_0=0 +datum=NAD83
+units=m +no_defs=True
nearest
(5.0, 5.0)
```



Other formats supported by rasterio, (e.g., PROJ4 strings) can be used.

```
with gw.config.update(ref_crs=proj4):
    with gw.open(rgbn) as src:
        print(src.transform)
        print(src.crs)
        print(src.resampling)
        print(src.res)
```

```
(5.0, 0.0, 2502400.7632678417, 0.0, -5.0, -2147313.7330151177)
+proj=aea +lat_0=40 +lon_0=-96 +lat_1=20 +lat_2=60 +x_0=0 +y_0=0 +datum=NAD83
+units=m +no_defs=True
nearest
(5.0, 5.0)
```

Resampling the Cell Size

The resampling algorithm can be specified in the `geowombat.open` function. Here, we use cubic convolution resampling to warp the data to EPSG code 31972 (a UTM projection).

```
with gw.config.update(ref_crs=31972):
    with gw.open(rgbn, resampling='cubic') as src:
        print(src.transform)
        print(src.crs)
        print(src.resampling)
        print(src.res)
```

```
(5.0, 0.0, 792988.0000004865, 0.0, -5.0, 2050381.9999358936)
+init=epsg:31972
cubic
(5.0, 5.0)
```

The transformed cell resolution can be added in the context manager. Here, we resample the data to 10m x 10m spatial resolution.

```
with gw.config.update(ref_crs=31972, ref_res=(10, 10)):
    with gw.open(rgbn, resampling='cubic') as src:
        print(src.transform)
        print(src.crs)
        print(src.resampling)
        print(src.res)
```

```
(10.0, 0.0, 792988.0000004865, 0.0, -10.0, 2050381.9999358936)
+init=epsg:31972
cubic
(10.0, 10.0)
```

Transformations Outside Context Manager

To transform an `xarray.DataArray` outside of a configuration context, use the `geowombat.transform_crs` function.

```
with gw.open(rgbn) as src:
    print(src.transform)
    print(src.crs)
    print(src.resampling)
    print(src.res)
    print('')
    src_tr = src.gw.transform_crs(proj4, dst_res=(10, 10), resampling='bilinear')
    print(src_tr.transform)
    print(src_tr.crs)
    print(src_tr.resampling)
    print(src_tr.res)
```

```
(5.0, 0.0, 792988.0, 0.0, -5.0, 2050382.0)
+init=epsg:32618
nearest
(5.0, 5.0)
```

```
(10.0, 0.0, 2502400.7632678417, 0.0, -10.0, -2147313.7330151177)
PROJCS["unknown",GEOGCS["unknown",DATUM["North_American_Datum_1983",SPHEROID["GRS
1980",6378137,298.257222101,AUTHORITY["EPSG","7019"]],AUTHORITY["EPSG","6269"]],PR
IMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],UNIT["degree",0.0174532925199433,AUTH
ORITY["EPSG","9122"]]],PROJECTION["Albers_Conic_Equal_Area"],PARAMETER["latitude_o
f_center",40],PARAMETER["longitude_of_center",-96],PARAMETER["standard_parallel_1"
,20],PARAMETER["standard_parallel_2",60],PARAMETER["false_easting",0],PARAMETER["f
alse_northing",0],UNIT["metre",1,AUTHORITY["EPSG","9001"]],AXIS["Easting",EAST],AX
IS["Northing",NORTH]]
bilinear
(10, 10)
```

For more help we can read through the docs a bit.

```
with gw.open(rgbn, resampling='cubic') as src:
    print(help(src.gw.transform_crs))
```

```
Help on method transform_crs in module geowombat.core.geoxarray:

transform_crs(dst_crs=None, dst_res=None, dst_width=None, dst_height=None,
dst_bounds=None, resampling='nearest', warp_mem_limit=512, num_threads=1) method
of geowombat.core.geoxarray.GeoWombatAccessor instance
    Transforms a DataArray to a new coordinate reference system

    Args:
        dst_crs (Optional[CRS | int | dict | str]): The destination CRS.
        dst_res (Optional[tuple]): The destination resolution.
        dst_width (Optional[int]): The destination width. Cannot be used with
``dst_res``.
        dst_height (Optional[int]): The destination height. Cannot be used with
``dst_res``.
        dst_bounds (Optional[BoundingBox | tuple]): The destination bounds, as a
``rasterio.coords.BoundingBox``
            or as a tuple of (left, bottom, right, top).
        resampling (Optional[str]): The resampling method if ``filename`` is a
``list``.
            Choices are ['average', 'bilinear', 'cubic', 'cubic_spline', 'gauss',
'lanczos', 'max', 'med', 'min', 'mode', 'nearest'].
        warp_mem_lim    it (Optional[int]): The warp memory limit.
        num_threads (Optional[int]): The number of parallel threads.

    Returns:
        ``xarray.DataArray``

    Example:
        >>> import geowombat as gw
        >>>
        >>> with gw.open('image.tif') as src:
        >>>     dst = src.gw.transform_crs(4326)

None
```

Learning Objectives

- Create mosaics of more than one multiband image
- Find the intersection of two images
- View the footprint of multiple image tiles

Review

- Opening Remotely Sensed Data
- Raster Operations

Handle Multiple Remotely Sensed Images

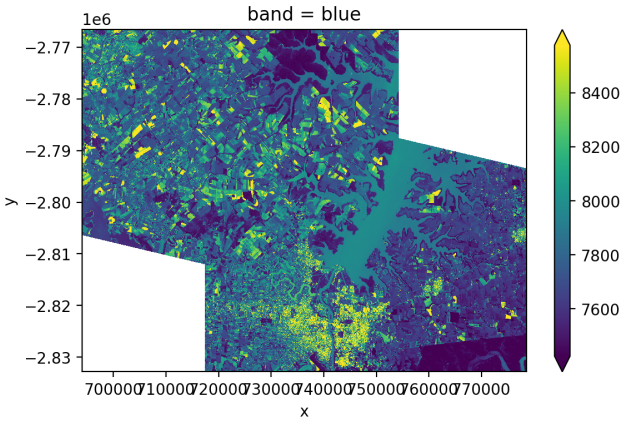
Doing analysis over larger areas often requires the use of image mosaics (combining two or more images). Luckily for us geowombat makes this process relatively easy.

Union (Mosaic) of Remotely Sensed Image

As an example let's plot the union with `mosaic=True` of two images taken on the same day, for the overlapping portions we will use the mean pixel value by setting `overlap='mean'`, but blue band only. Alternatively we could use one of 'mean', 'min', or 'max'.

Note we rename the band name with `band_names=['blue']`.

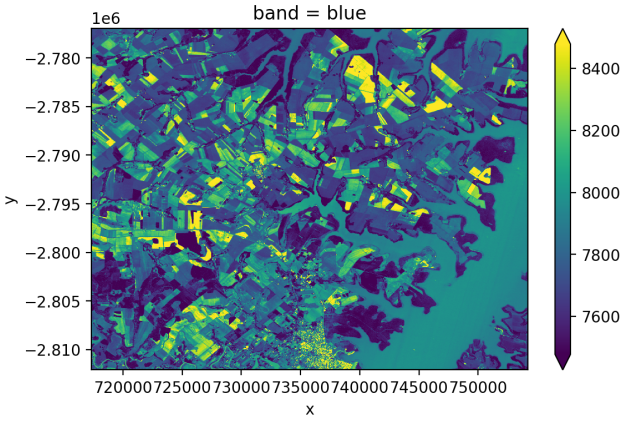
```
# Import GeoWombat
import geowombat as gw

# import plotting
import matplotlib.pyplot as plt
import matplotlib.patheffects as pe

# load data
from geowombat.data import l8_224077_20200518_B2, l8_224078_20200518_B2

fig, ax = plt.subplots(dpi=200)
filenames = [l8_224077_20200518_B2, l8_224078_20200518_B2]
with gw.open(filenames,
                band_names=['blue'],
                mosaic=True,
                overlap = 'mean',
                bounds_by='union') as src:
    src.where(src != 0).sel(band='blue').plot.imshow(robust=True, ax=ax)
plt.tight_layout(pad=1)
```



Intersection of Remotely Sensed Image

Same idea with the intersection, using `bounds_by='intersection'`, we still need to mosaic the two images `mosaic=True`.

```
fig, ax = plt.subplots(dpi=200)
filenames = [l8_224077_20200518_B2, l8_224078_20200518_B2]
with gw.open(filenames,
                band_names=['blue'],
                mosaic=True,
                overlap='mean',
                bounds_by='intersection') as src:
    src.where(src != 0).sel(band='blue').plot.imshow(robust=True, ax=ax)
plt.tight_layout(pad=1)
```



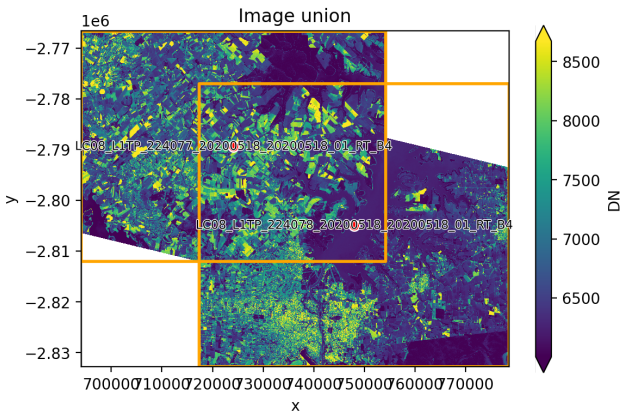
View Image Tile Footprints

Here we set up a more complicated plotting function for near IR 'nir'. Note the use of `footprint_grid`.

```
from geowombat.data import l8_224077_20200518_B4, l8_224078_20200518_B4

def plot(bounds_by, ref_image=None, cmap='viridis'):
    fig, ax = plt.subplots(dpi=200)
    with gw.config.update(ref_image=ref_image):
        with gw.open([l8_224077_20200518_B4, l8_224078_20200518_B4],
                     band_names=['nir'],
                     chunks=256,
                     mosaic=True,
                     bounds_by=bounds_by) as srca:
            # Plot the NIR band
            srca.where(srca != 0).sel(band='nir').plot.imshow(robust=True,
cbar_kwargs={'label': 'DN'}, ax=ax)
            # Plot the image chunks
            srca.gw.chunk_grid.plot(color='none', edgecolor='k', ls='-', lw=0.5,
ax=ax)
            # Plot the image footprints
            srca.gw.footprint_grid.plot(color='none', edgecolor='orange', lw=2,
ax=ax)
            # Label the image footprints
            for row in srca.gw.footprint_grid.itertuples(index=False):
                ax.scatter(row.geometry.centroid.x, row.geometry.centroid.y,
                           s=50, color='red', edgecolor='white', lw=1)
                ax.annotate(row.footprint.replace('.TIF', ''),
                           (row.geometry.centroid.x, row.geometry.centroid.y),
                           color='black',
                           size=8,
                           ha='center',
                           va='center',
                           path_effects=[pe.withStroke(linewidth=1,
foreground='white')])
            # Set the display bounds
            ax.set_ylim(srca.gw.footprint_grid.total_bounds[1]-10,
srca.gw.footprint_grid.total_bounds[3]+10)
            ax.set_xlim(srca.gw.footprint_grid.total_bounds[0]-10,
srca.gw.footprint_grid.total_bounds[2]+10)
    title = f'Image {bounds_by}' if bounds_by else
str(Path(ref_image).name.split('.')[0]) + ' as reference'
    size = 12 if bounds_by else 8
    ax.set_title(title, size=size)
    plt.tight_layout(pad=1)
```

The two plots below illustrate how two images can be mosaicked. The orange grids highlight the image footprints while the black grids illustrate the `DataArray` chunks.

```
plot('union')
```



Learning Objectives

- Learn about basic principals of band math
- Calculate common indicies like NDVI, EVI etc

Review

- Raster Data
- Opening Remotely Sensed Data
- Sensor specific configurations

Band Math & Vegetation Indices

Vegetation indices

Healthy vegetation (with chlorophyll) reflects more near-infrared (NIR) and green light compared to other wavelengths and absorbs more red and blue light. We can use this effect to generate a number of vegetation indices including the following:

Enhanced Vegetation Index (EVI)

EVI is an index of vegetation that is optimized to improve sensitivity to high biomass and better handling of background and atmospheric influences. It is calculated with the formula below using the Near Infrared (NIR), Red and Blue bands. There are also a number of parameters like C_1 that are specific to each sensor. Luckily geowombat handles this all for you!

$$EVI = G \times \frac{NIR - Red}{NIR + C_1 \times Red - C_2 \times Blue + L}$$

The result of this formula generates a value between -1 and +1. Low reflectance (low values) in the red channel and high reflectance in the NIR channel will yield a high EVI value.

```
import geowombat as gw
from geowombat.data import rgbn
import matplotlib.pyplot as plt
```

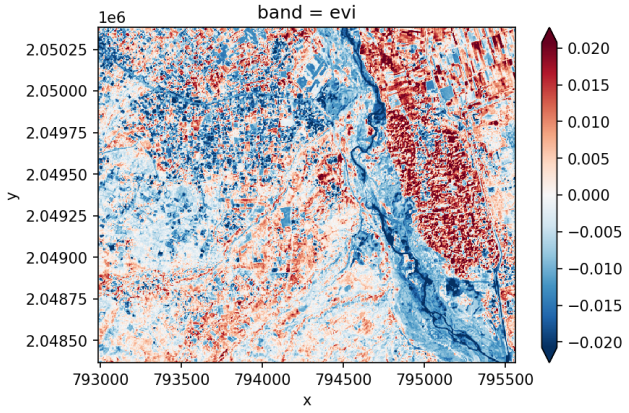
Calculate a vegetation index, returning an `Xarray.DataArray`.

```
with gw.open(rgbn) as ds:
    print(ds)
    evi = ds.gw.evi(sensor='rgbn', scale_factor=0.0001)
    print(evi)
```

```
<xarray.DataArray (band: 4, y: 403, x: 515)>
dask.array<open_rasterio-b3d223d294244e0149d01d15b4a9c936<this-array>, shape=(4,
403, 515), dtype=uint8, chunksize=(1, 64, 64), chunktype=numpy.ndarray>
Coordinates:
  * band     (band) int64 1 2 3 4
  * y        (y) float64 2.05e+06 2.05e+06 2.05e+06 ... 2.048e+06 2.048e+06
  * x        (x) float64 7.93e+05 7.93e+05 7.93e+05 ... 7.956e+05 7.956e+05
Attributes: (12/13)
    transform:          (5.0, 0.0, 792988.0, 0.0, -5.0, 2050382.0)
    crs:                +init=epsg:32618
    res:                (5.0, 5.0)
    is_tiled:           1
    nodatavals:         (nan, nan, nan, nan)
    scales:             (1.0, 1.0, 1.0, 1.0)
    ...                 ...
    AREA_OR_POINT:      Area
    DataType:           Generic
    filename:           /home/mmann1123/anaconda3/envs/pygisbookgw/lib/python...
    resampling:         nearest
    data_are_separate:  0
    data_are_stacked:   0
<xarray.DataArray (band: 1, y: 403, x: 515)>
dask.array<broadcast_to, shape=(1, 403, 515), dtype=float64, chunksize=(1, 64,
64), chunktype=numpy.ndarray>
Coordinates:
  * y        (y) float64 2.05e+06 2.05e+06 2.05e+06 ... 2.048e+06 2.048e+06
  * x        (x) float64 7.93e+05 7.93e+05 7.93e+05 ... 7.956e+05 7.956e+05
  * band     (band) <U3 'evi'
Attributes: (12/17)
    transform:          (5.0, 0.0, 792988.0, 0.0, -5.0, 2050382.0)
    crs:                +init=epsg:32618
    res:                (5.0, 5.0)
    is_tiled:           1
    nodatavals:         None
    scales:             (1.0,)
    ...                 ...
    data_are_separate:  0
    data_are_stacked:   0
    pre-scaling:        0.0001
    sensor:             rgbn
    vi:                 evi
    drange:             (0, 1)
```

Or use the configuration context to set parameters.

```
fig, ax = plt.subplots(dpi=150)

with gw.config.update(sensor='rgbn', scale_factor=0.0001):
    with gw.open(rgbn) as ds:
        evi = ds.gw.evi()
        evi.plot(robust=True, ax=ax)
plt.tight_layout(pad=1)
```

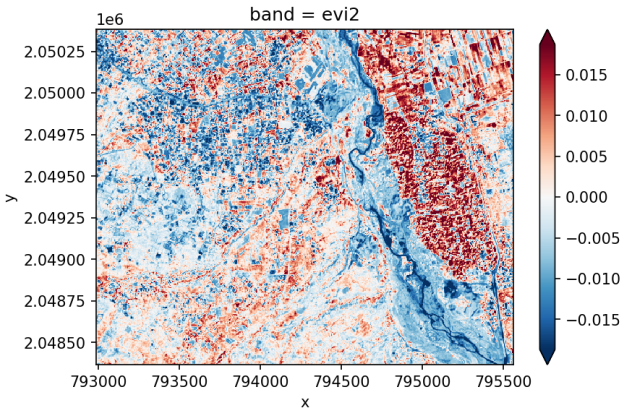


Two-band Enhanced Vegetation Index (EVI2)

We can also calcuate an approximation of EVI with two bands using $G \times ((NIR - RED)/(L + NIR + C \times Red))$

This allows us to extend EVI calculations back in time using AVHRR, and avoids some problems with the blue band which tends to be noisy.

```
fig, ax = plt.subplots(dpi=150)

with gw.config.update(sensor='rgbn', scale_factor=0.0001):
    with gw.open(rgbn) as ds:
        evi2 = ds.gw.evi2()
        evi2.plot(robust=True, ax=ax)
plt.tight_layout(pad=1)
```



Normalized Difference Indices (NDVI)

The simplest vegetation metric is NDVI, which is just the normalized difference between the Red and NIR bands. It is calculated as follows $\frac{NIR-Red}{NIR+Red}$.

We can calculate it using the generic `norm_diff` function for any two-band combination.

```
with gw.config.update(sensor='rgbn'):
    with gw.open(rgbn) as ds:
        d = ds.gw.norm_diff('red', 'nir')
        print(d)
```

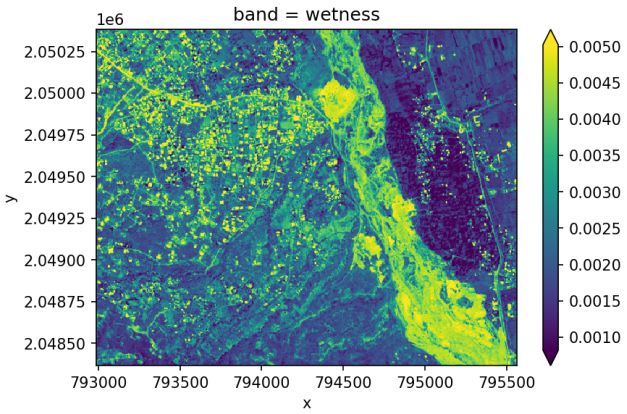
```
<xarray.DataArray (band: 1, y: 403, x: 515)>
dask.array<broadcast_to, shape=(1, 403, 515), dtype=float64, chunksize=(1, 64,
64), chunktype=numpy.ndarray>
Coordinates:
  * y        (y) float64 2.05e+06 2.05e+06 2.05e+06 ... 2.048e+06 2.048e+06
  * x        (x) float64 7.93e+05 7.93e+05 7.93e+05 ... 7.956e+05 7.956e+05
  * band     (band) <U9 'norm-diff'
Attributes: (12/17)
    transform:          (5.0, 0.0, 792988.0, 0.0, -5.0, 2050382.0)
    crs:                +init=epsg:32618
    res:                (5.0, 5.0)
    is_tiled:           1
    nodatavals:         None
    scales:             (1.0,)
    ...                 ...
    resampling:         nearest
    data_are_separate:  0
    data_are_stacked:   0
    pre-scaling:        1.0
    vi:                 norm-diff
    drange:             (-1, 1)
```

Tasseled Cap Transformations

Tasseled cap tranform uses a linear equation to try to differentiate different components of the spectrum that are of interest for vegetation dynamics such as phenological stages. The output includes three bands including `brightness`, `greeness` for vegetation, and `wetness` as an idicator of soil and canopy moisture. Use `.sel(band='wetness')` to select them individually.

```
fig, ax = plt.subplots(dpi=150)

with gw.config.update(sensor='qb', scale_factor=0.0001):
    with gw.open(rgbn) as ds:
        tcap = ds.gw.tasseled_cap()
        tcap.sel(band='wetness').plot(robust=True, ax=ax)
        print(tcap)
plt.tight_layout(pad=1)
```

```
<xarray.DataArray (band: 3, y: 403, x: 515)>
dask.array<astype, shape=(3, 403, 515), dtype=float64, chunksize=(3, 64, 64),
chunktype=numpy.ndarray>
Coordinates:
  * y        (y) float64 2.05e+06 2.05e+06 2.05e+06 ... 2.048e+06 2.048e+06
  * x        (x) float64 7.93e+05 7.93e+05 7.93e+05 ... 7.956e+05 7.956e+05
  * band     (band) <U10 'brightness' 'greenness' 'wetness'
Attributes: (12/14)
    transform:          (5.0, 0.0, 792988.0, 0.0, -5.0, 2050382.0)
    crs:                +init=epsg:32618
    res:                (5.0, 5.0)
    is_tiled:           1
    nodatavals:         (nan, nan, nan, nan)
    scales:             (1.0, 1.0, 1.0, 1.0)
    ...                 ...
    DataType:           Generic
    sensor:             Quickbird with 4 (visible + NIR) bands
    filename:           /home/mmann1123/anaconda3/envs/pygisbookgw/lib/python...
    resampling:         nearest
    data_are_separate:  0
    data_are_stacked:   0
```



Sources:

- Wikipedia EVI
- Wikipedia Tasseled Cap

Learning Objectives

- Subset bands by index or name
- Extract raster data by row and column number
- Extract data by bounding window
- Extract raster data by coordinates
- Extract raster data by geometry (point, polygon)

Review

- Data Structures
- Raster Data
- Reading and writing remotely sensed data

Raster Data Extraction

Raster data is often of little use unless we can extract and summarize the data. For instance, extracting raster to points by coordinates allows us to pass data to machine learning models for land cover classification or cloud masking.

Subsetting rasters

We can subset sections of the data array in a number of ways. In this case we will create a slice based on row and column location to subset LandSat data using a `rasterio.window.Window`.

Either a `rasterio.window.Window` object or tuple can be used with `geowombat.open`.

```
import geowombat as gw
from geowombat.data import rgbn

from rasterio.windows import Window
w = Window(row_off=0, col_off=0, height=100, width=100)

with gw.open(rgbn,
                band_names=['blue', 'green', 'red'],
                num_workers=8,
                indexes=[1, 2, 3],
                window=w,
                out_dtype='float32') as src:
    print(src)
```

```
<xarray.DataArray 'array-c1686f70048cb766e2fe94fb3c34fb6d' (band: 3, y: 100, x:
100)>
dask.array<array, shape=(3, 100, 100), dtype=float32, chunksize=(1, 64, 64),
chunktype=numpy.ndarray>
Coordinates:
  * band     (band) <U5 'blue' 'green' 'red'
  * y        (y) float64 2.05e+06 2.05e+06 2.05e+06 ... 2.05e+06 2.05e+06
  * x        (x) float64 7.93e+05 7.93e+05 7.93e+05 ... 7.935e+05 7.935e+05
Attributes:
    transform:          | 5.00, 0.00, 792988.00|\n| 0.00,-5.00, 2050382.00|\n...
    crs:                +init=epsg:32618
    res:                (5.0, 5.0)
    is_tiled:           1
    nodatavals:         (nan, nan, nan, nan)
    offsets:            (0.0, 0.0, 0.0, 0.0)
    data_are_separate:  0
    data_are_stacked:   0
```

We can also slice a subset of data using a tuple of bounded coordinates.

```
bounds = (793475.76, 2049033.03, 794222.03, 2049527.24)

with gw.open(rgbn,
                band_names=['green', 'red', 'nir'],
                num_workers=8,
                indexes=[2, 3, 4],
                bounds=bounds,
                out_dtype='float32') as src:
    print(src)
```

The configuration manager provides an alternative method to subset rasters. See `tutorial-config` for more details.

```
with gw.config.update(ref_bounds=bounds):

    with gw.open(rgbn) as src:
        print(src)
```

By default, the subset will be returned by the upper left coordinates of the bounds, potentially shifting cell alignment with the reference raster. To subset a raster and align it to the same grid, use the **ref_tar** keyword. This is equivalent to a “snap raster” in ArcGIS.

```
with gw.config.update(ref_bounds=bounds, ref_tar=rgbn):

    with gw.open(rgbn) as src:
        print(src)
```

Extracting data by coordinates

To extract values at a coordinate pair, translate the coordinates into array indices. For extraction by geometry, for instance with a shapefile, see extract by point geometry.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518

# Coordinates in map projection units
y, x = -2823031.15, 761592.60

with gw.open(l8_224078_20200518) as src:
    # Transform the map coordinates to data indices
    j, i = gw.coords_to_indices(x, y, src)
    # Subset by index
    data = src[:, i, j].data.compute()

print(data.flatten())
```

```
[7448 6882 6090]
```

A latitude/longitude pair can be extracted after converting to the map projection.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518

# Coordinates in latitude/longitude
lat, lon = -25.50142964, -54.39756038

with gw.open(l8_224078_20200518) as src:
    # Transform the coordinates to map units
    x, y = gw.lonlat_to_xy(lon, lat, src)
    # Transform the map coordinates to data indices
    j, i = gw.coords_to_indices(x, y, src)
    data = src[:, i, j].data.compute()

print(data.flatten())
```

```
[7448 6882 6090]
```

Extracting data with point geometry

In the example below, ‘l8_224078_20200518_points’ is a GeoPackage of point locations, and the output `df` is a GeoPandas GeoDataFrame. To extract the raster values at the point locations, use `geowombat.extract`.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518, l8_224078_20200518_points

with gw.open(l8_224078_20200518) as src:
    df = src.gw.extract(l8_224078_20200518_points)

print(df)
```

```
        name                        geometry  id     1     2     3
0      water  POINT (741522.314 -2811204.698)   0  7966  7326  6254
1       crop  POINT (736140.845 -2806478.364)   1  8030  7490  8080
2       tree  POINT (745919.508 -2805168.579)   2  7561  6874  6106
3  developed  POINT (739056.735 -2811710.662)   3  8302  8202  8111
4      water  POINT (737802.183 -2818016.412)   4  8277  7982  7341
5       tree  POINT (759209.443 -2828566.230)   5  7398  6711  6007
```

> **Note**
>
> The line `df = src.gw.extract(l8_224078_20200518_points)` could also have been written as `df = gw.extract(src, l8_224078_20200518_points)`.

In the previous example, the point vector had a CRS that matched the raster (i.e., EPSG=32621, or UTM zone 21N). If the CRS had not matched, the `geowombat.extract` function transforms the CRS on-the-fly.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518, l8_224078_20200518_points
import geopandas as gpd

point_df = gpd.read_file(l8_224078_20200518_points)
print(point_df.crs)

# Transform the CRS to WGS84 lat/lon
point_df = point_df.to_crs('epsg:4326')
print(point_df.crs)

with gw.open(l8_224078_20200518) as src:
    df = src.gw.extract(point_df)

print(df)
```

```
epsg:32621
epsg:4326
```

```
       name                        geometry  id     1     2     3
0     water  POINT (741522.314 -2811204.698)   0  7966  7326  6254
1      crop  POINT (736140.845 -2806478.364)   1  8030  7490  8080
2      tree  POINT (745919.508 -2805168.579)   2  7561  6874  6106
3  developed  POINT (739056.735 -2811710.662)   3  8302  8202  8111
4     water  POINT (737802.183 -2818016.412)   4  8277  7982  7341
5      tree  POINT (759209.443 -2828566.230)   5  7398  6711  6007
```

Set the data band names using `sensor = 'bgr'`, which assigns the band names blue, green, red.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518, l8_224078_20200518_points

with gw.config.update(sensor='bgr'):
    with gw.open(l8_224078_20200518) as src:
        df = src.gw.extract(l8_224078_20200518_points,
                            band_names=src.band.values.tolist())

print(df)
```

```
       name                        geometry  id  blue  green   red
0     water  POINT (741522.314 -2811204.698)   0  7966   7326  6254
1      crop  POINT (736140.845 -2806478.364)   1  8030   7490  8080
2      tree  POINT (745919.508 -2805168.579)   2  7561   6874  6106
3  developed  POINT (739056.735 -2811710.662)   3  8302   8202  8111
4     water  POINT (737802.183 -2818016.412)   4  8277   7982  7341
5      tree  POINT (759209.443 -2828566.230)   5  7398   6711  6007
```

Extracting time series images by point geometry

We can also easily extract a time series of raster images. Extracted pixel values are provided in 'wide' format with appropriate labels, for instance the column 't2_blue' would be the blue band for the second time period

```
from geowombat.data import l8_224078_20200518, l8_224078_20200518_points

with gw.config.update(sensor='bgr'):
    with gw.open([l8_224078_20200518, l8_224078_20200518],
            time_names=['t1', 't2'],
            stack_dim='time') as src:

        # Extract and by point geometry
        df = src.gw.extract(l8_224078_20200518_points)

print(df)
```

```
       name                        geometry  id  t1_blue  t1_green  t1_red  \
0     water  POINT (741522.314 -2811204.698)   0     7966      7326    6254
1      crop  POINT (736140.845 -2806478.364)   1     8030      7490    8080
2      tree  POINT (745919.508 -2805168.579)   2     7561      6874    6106
3  developed  POINT (739056.735 -2811710.662)   3     8302      8202    8111
4     water  POINT (737802.183 -2818016.412)   4     8277      7982    7341
5      tree  POINT (759209.443 -2828566.230)   5     7398      6711    6007

   t2_blue  t2_green  t2_red
0     7966      7326    6254
1     8030      7490    8080
2     7561      6874    6106
3     8302      8202    8111
4     8277      7982    7341
5     7398      6711    6007
```

Extracting data by polygon geometry

To extract values within polygons, use the same `geowombat.extract` function.

```
from geowombat.data import l8_224078_20200518, l8_224078_20200518_polygons

with gw.config.update(sensor='bgr'):
    with gw.open(l8_224078_20200518) as src:
        df = src.gw.extract(l8_224078_20200518_polygons,
                            band_names=src.band.values.tolist())

    print(df)
```

```
     id  point                          geometry       name   blue  green  \
0     0      0  POINT (737535.000 -2795205.000)      water   8017   7435
1     0      1  POINT (737565.000 -2795205.000)      water   8016   7439
2     0      2  POINT (737595.000 -2795205.000)      water   8012   7442
3     0      3  POINT (737625.000 -2795205.000)      water   7997   7422
4     0      4  POINT (737655.000 -2795205.000)      water   7997   7405
..   ..    ...                             ...        ...    ...    ...
667   3    667  POINT (739005.000 -2811795.000)  developed   9014   8236
668   3    668  POINT (739035.000 -2811795.000)  developed   8567   8564
669   3    669  POINT (739065.000 -2811795.000)  developed   8099   7676
670   3    670  POINT (739095.000 -2811795.000)  developed  10151   9651
671   3    671  POINT (739125.000 -2811795.000)  developed   8065   7735

       red
0     6283
1     6294
2     6295
3     6284
4     6266
..     ...
667   8325
668   8447
669   7332
670  10153
671   7501

[672 rows x 7 columns]
```

Calculate mean pixel value by polygon

It is simple then to calculate the mean value of pixels within each polygon by using the polygon `id` column and pandas groupby function. You can easily calculate other statistics like min, max, median etc.

```
from geowombat.data import l8_224078_20200518, l8_224078_20200518_polygons

with gw.config.update(sensor='bgr'):
    with gw.open(l8_224078_20200518) as src:
        df = src.gw.extract(l8_224078_20200518_polygons,
                            band_names=src.band.values.tolist())
        # use pandas groupby to calc pixel mean
        df = df.groupby('id').mean()
    print(df)
```

```
    point         blue        green          red
id
0   103.5  7990.052885  7388.432692  6264.807692
1   304.0  7692.481865  7037.419689  7571.207254
2   497.0  7506.901554  6838.704663  6091.932642
3   632.5  8668.423077  8294.717949  8312.192308
```

Learning Objectives

- Fit and predict machine learning models to make spatial predictions
  - Use sklearn pipelines, cross-validation and hyper parameter tuning for spatial data
- Predict landcover or continuous models
- Make predictions using timeseries data

Review

- Geowombat IO
- Geowombat Extraction
- Sklearn_xarray
- Sklearn pipelines

Spatial Prediction using ML in Python

Create Land Use Classification using Geowombat & Sklearn

The most common task for remotely sensed data is creating land cover classification. In this tutorial we will walk you through how to train a ML model using raster data. These methods are heavily dependent on the great package sklearn_xarray. To understand the pipeline commands please see their documentation and examples.

Supervised Classification in Python

In the following example we will use Landsat data, some training data to train a supervised sklearn model. In order to do this we first need to have land classifications for a set of points of polygons. In this case we have three polygons with the classes ['water','crop','tree','developed']. The first step is to use `LabelEncoder` to convert these to integer based categories, which we store in a new column called 'lc'.

```
import geowombat as gw
from geowombat.data import l8_224078_20200518, l8_224078_20200518_polygons
from geowombat.ml import fit, predict, fit_predict
import geopandas as gpd
from sklearn_xarray.preprocessing import Featurizer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.decomposition import PCA
from sklearn.naive_bayes import GaussianNB

le = LabelEncoder()

# The labels are string names, so here we convert them to integers
labels = gpd.read_file(l8_224078_20200518_polygons)
labels['lc'] = le.fit(labels.name).transform(labels.name)
print(labels)
```

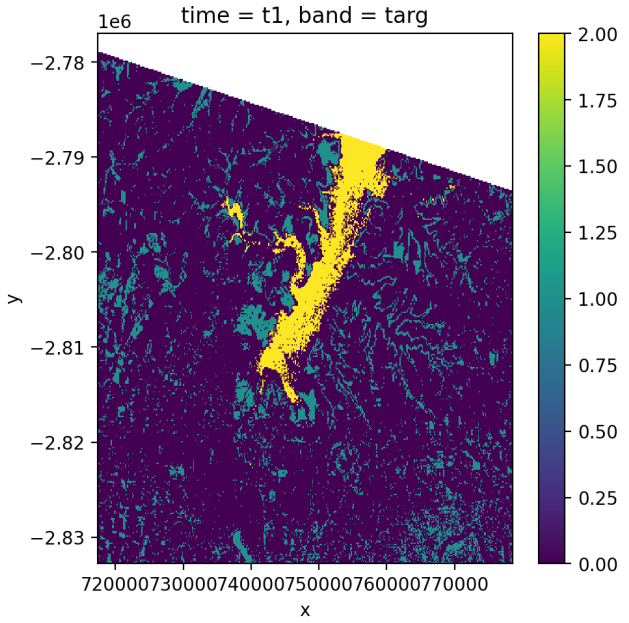
```
        name                                           geometry  lc
0      water  POLYGON ((737544.502 -2795232.772, 737544.502 ...   3
1       crop  POLYGON ((742517.658 -2798160.232, 743046.717 ...   0
2       tree  POLYGON ((742435.360 -2801875.403, 742458.874 ...   2
3  developed  POLYGON ((738903.667 -2811573.845, 738926.586 ...   1
```

We are then going to generate our sklearn pipeline (see simple tutorial here). A pipeline simply allows us to pass a numpy array through a defined set of operations. In this case the data is passed through the following operations:

- `StandardScaler`: Normalizes all variables by removing the mean and scaling to unit variance
- `PCA`: Calculates Principal Components to reduce dimensionality.
- `GaussianNB`: Fits a Gaussian Naive Bayes model for a quick classification.

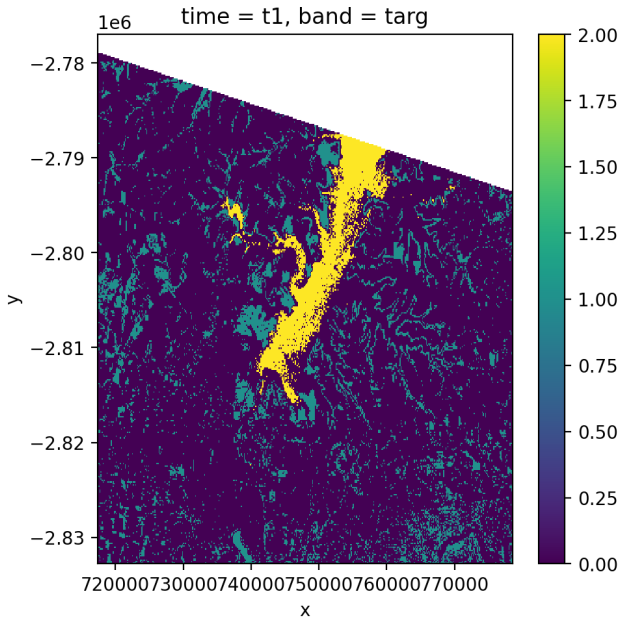
In this example we will fit and predict the model in two steps. The `fit` method returns three objects, a transformed version of the original dataset `X` that can be used by sklearn, `Xy` a tuple containing the data used for training `(X,y)` where any data outside the polygons is removed, and the trained pipeline `clf` object.

```
import matplotlib.pyplot as plt

# Use a data pipeline
pl = Pipeline([ ('scaler', StandardScaler()),
                ('pca', PCA()),
                ('clf', GaussianNB())])

fig, ax = plt.subplots(dpi=200,figsize=(5,5))

# Fit the classifier
with gw.config.update(ref_res=150):
    with gw.open(l8_224078_20200518) as src:
        X, Xy, clf = fit(src, pl, labels, col="lc")
        y = predict(src, X, clf)
        y.plot(robust=True, ax=ax)
plt.tight_layout(pad=1)
```



In order to fit and predict to our original data in one step, we simply use `fit_predict`:

```
from geowombat.ml import fit_predict
fig, ax = plt.subplots(dpi=200,figsize=(5,5))

with gw.config.update(ref_res=150):
    with gw.open(l8_224078_20200518) as src:
        y = fit_predict(src, pl, labels, col='lc')
        y.plot(robust=True, ax=ax)
plt.tight_layout(pad=1)
```

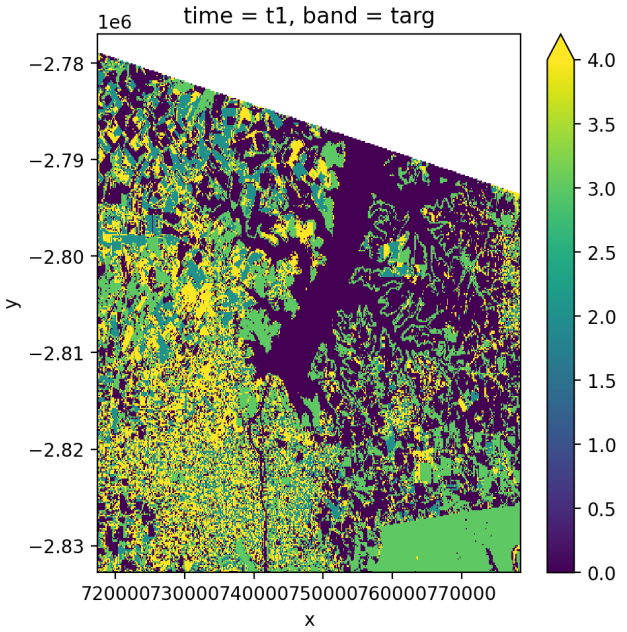


Unsupervised Classification in Python

Unsupervised classification takes a different approach. Here we don't have to provide examples of different land cover types. Instead we rely on the algorithm to identify distinct clusters of similar data, and apply a unique label to each cluster. For instance, if we are talking about land cover water and trees are going to look very different. Water reflects more blue and absorbs all the near infrared, while trees reflect little blue and reflect lots of near infrared. Therefore water and trees should 'cluster' together when plotted out according to their different blue and near infrared reflectances. These clusters will be assigned a unique value to each pixel, e.g. water will be assigned 1 and trees 2. Later, the end user will need to go back and assign the label to each numbered cluster, e.g. water=1, trees=2.

In this example we will use kmeans to do our clustering. To run we need to decide apriori how many clusters we want to identify. Typically you want to roughly double the number of expected classes and then recombine them later into the desired labels. This helps to better understand and categorize the variation in your image.

```
from sklearn.cluster import KMeans
cl = Pipeline([ ('clf', KMeans(n_clusters=6, random_state=0))])

fig, ax = plt.subplots(dpi=200,figsize=(5,5))

# Fit_predict unsupervised classifier
with gw.config.update(ref_res=150):
    with gw.open(l8_224078_20200518) as src:
        y= fit_predict(src, cl)
        y.plot(robust=True, ax=ax)
plt.tight_layout(pad=1)
```



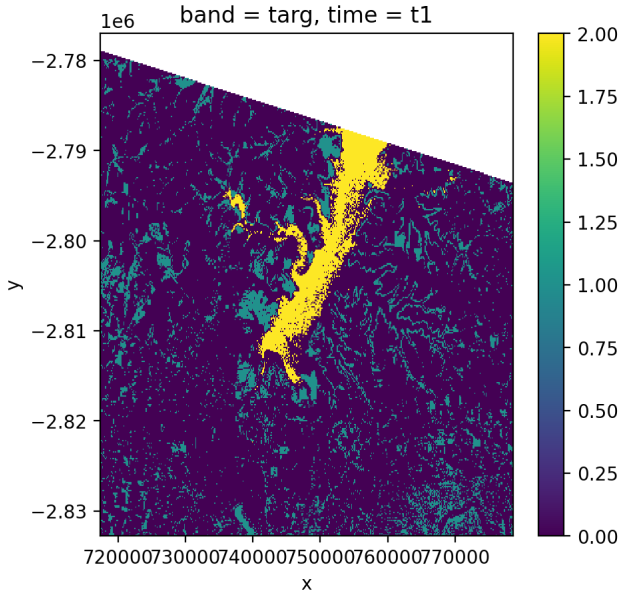
In this case we can see that it effective labels different clusters of data, and now it is up to us to determine which clusters should be categorized as water, trees, and fields etc.

Spatial prediction with time series stack using Geowombat & Sklearn

If you have a stack of time series data it is simple to apply the same method as we described previously, except we need to open multiple images, set `stack_dim` to 'time' and set the `time_names`. *Note* we are just pretending we have two dates of LandSat imagery here.

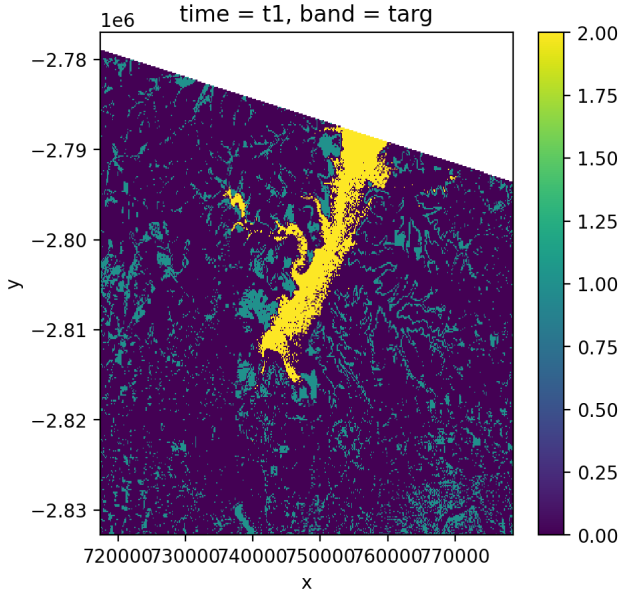
```
fig, ax = plt.subplots(dpi=200,figsize=(5,5))

with gw.config.update(ref_res=150):
   with gw.open([l8_224078_20200518, l8_224078_20200518], time_names=['t1', 't2'],
stack_dim='time') as src:
        y = fit_predict(src, pl, labels, col='lc')
        print(y)
        # plot one time period prediction
        y.sel(time='t1').plot(robust=True, ax=ax)
```

```
<xarray.DataArray (time: 2, y: 372, x: 408)>
dask.array<getitem, shape=(2, 372, 408), dtype=float64, chunksize=(1, 256, 256),
chunktype=numpy.ndarray>
Coordinates:
    band     <U4 'targ'
  * y        (y) float64 -2.777e+06 -2.777e+06 ... -2.833e+06 -2.833e+06
  * x        (x) float64 7.174e+05 7.176e+05 7.177e+05 ... 7.783e+05 7.785e+05
  * time     (time) <U2 't1' 't2'
    targ     (time, y, x) float64 nan nan nan nan nan ... nan nan nan nan nan
Attributes:
    transform:          (150.0, 0.0, 717345.0, 0.0, -150.0, -2776995.0)
    crs:                +init=epsg:32621
    res:                (150.0, 150.0)
    is_tiled:           0
    nodatavals:         (0, 0, 0)
    scales:             (1.0, 1.0, 1.0)
    offsets:            (0.0, 0.0, 0.0)
    filename:           ['LC08_L1TP_224078_20200518_20200518_01_RT.TIF', 'LC0...
    resampling:         nearest
    AREA_OR_POINT:      Area
    data_are_separate:  1
    data_are_stacked:   1
```



If you want to do more sophisticated model tuning using sklearn it is also possible to break up your fit and predict steps as follows:

```
fig, ax = plt.subplots(dpi=200,figsize=(5,5))

with gw.config.update(ref_res=150):
    with gw.open(l8_224078_20200518) as src:
        X, Xy, clf = fit(src, pl, labels, col="lc")
        y = predict(src, X, clf)
        y.plot(robust=True, ax=ax)
```



Cross-validation and Hyperparameter Tuning with Spatial Prediction

One of the most important parts of successfully building a model is a careful assessment of model performance. To do this we will leverage some of `sklearn` built in tools. One of the most common cross-validation methods is called k-fold, where you data is broken in to independent sets of training and testing data multiple times. The ability of the model - trained on the 'training' data - to predict the outcome of the 'testing' data multiple times. We can then have a measure of how well our model will work on data it has never seen before.

In this case we are going to use our supervised classification pipeline `pl` from earlier. And we will use kfold to do cross-validation. To use `kfold` with `geowombat` we need to use `CrossValidatorWrapper` as seen in the example below to allow it to work with `xarray` objects.

We often also need to hyper-parameter tune our model. In this case we will see if we need to keep 1, 2, or 3 pca components. We might also want to experiment with whether scaling the data range impacts our perforamnce with StandardScaler by changing whether or not variables are divided by their standard deviation.

To do hyper-parameter tuning with GridSearchCV in a pipeline we need to set up the 'parameter-grid'. This part can be a little confusing. To help us let's isolate the `Pipeline` and `param_grid` from the example below:

```
pl = Pipeline([('scaler', StandardScaler()),
               ('pca', PCA()),
               ('clf', GaussianNB())])

param_grid={"scaler__with_std":[True,False],
            "pca__n_components": [1, 2, 3]
            }
```

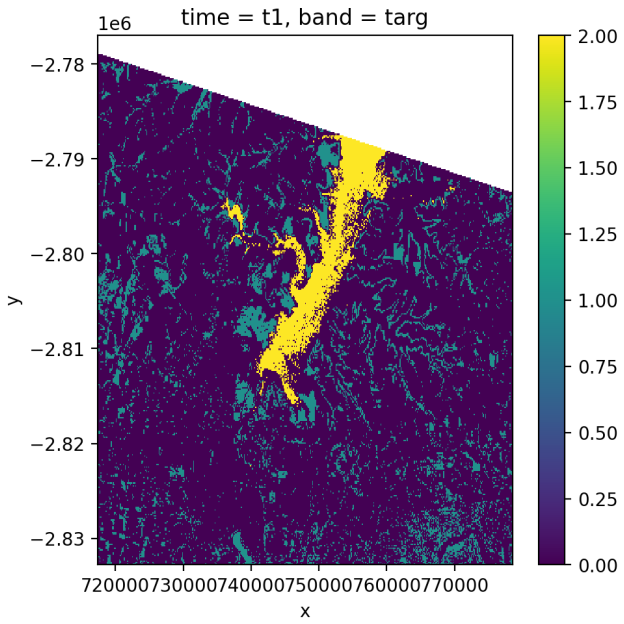
Notice that each step in the pipeline is labeled (e.g. 'scaler', 'pca', 'clf'). To try out different parameters for each step we are going to need to reference them by name in our `param_grid` dictionary. The dictionary follows this convention:

```
(step_name)__(parameter_name):[value_1, value2]
```

So `"pca__n_components": [1, 2, 3]` says that for the `pca` step of the pipeline, we will try out tree different values for the parameter `n_components`, allowing us to choose the one that performs best at predicting our 'testing' data.

```
from sklearn.model_selection import GridSearchCV, KFold
from sklearn_xarray.model_selection import CrossValidatorWrapper

pl = Pipeline([('scaler', StandardScaler()),
               ('pca', PCA()),
               ('clf', GaussianNB())])

cv = CrossValidatorWrapper(KFold())
gridsearch = GridSearchCV(pl, cv=cv, scoring='balanced_accuracy',
                    param_grid={
                      "scaler__with_std":[True,False],
                      "pca__n_components": [1, 2, 3]
                      })

fig, ax = plt.subplots(dpi=200,figsize=(5,5))

with gw.config.update(ref_res=150):
    with gw.open(l8_224078_20200518) as src:
        # fit a model to get Xy used to train model
        X, Xy, pipe = fit(src, pl, labels, col="lc")

        # fit cross valiation and parameter tuning
        # NOTE: must unpack * object Xy
        gridsearch.fit(*Xy)
        print(gridsearch.cv_results_)
        print(gridsearch.best_score_)
        print(gridsearch.best_params_)

        # get set tuned parameters and make the prediction
        # Note: predict(gridsearch.best_model_) not currently supported
        pipe.set_params(**gridsearch.best_params_)
        y = predict(src, X, pipe)
        y.plot(robust=True, ax=ax)
plt.tight_layout(pad=1)
```

```
{'mean_fit_time': array([0.15127478, 0.17484608, 0.14793749, 0.14352918,
0.15261936,
       0.18515682]), 'std_fit_time': array([0.00583673, 0.04766097, 0.00490675,
0.00348462, 0.00864314,
       0.03577372]), 'mean_score_time': array([0.09150372, 0.11217704, 0.09275851,
0.09279857, 0.10127025,
       0.11805472]), 'std_score_time': array([0.00606398, 0.02464995, 0.0030899 ,
0.00282972, 0.01786448,
       0.0282516 ]), 'param_pca__n_components': masked_array(data=[1, 1, 2, 2, 3,
3],
             mask=[False, False, False, False, False, False],
       fill_value='?',
            dtype=object), 'param_scaler__with_std': masked_array(data=[True,
False, True, False, True, False],
             mask=[False, False, False, False, False, False],
       fill_value='?',
            dtype=object), 'params': [{'pca__n_components': 1, 'scaler__with_std':
True}, {'pca__n_components': 1, 'scaler__with_std': False}, {'pca__n_components':
2, 'scaler__with_std': True}, {'pca__n_components': 2, 'scaler__with_std': False},
{'pca__n_components': 3, 'scaler__with_std': True}, {'pca__n_components': 3,
'scaler__with_std': False}], 'split0_test_score': array([1., 1., 1., 1., 1., 1.]),
'split1_test_score': array([1., 1., 1., 1., 1., 1.]), 'split2_test_score':
array([0.75 , 0.875, 0.75 , 0.75 , 0.625, 0.5  ]), 'split3_test_score': array([1.,
1., 1., 1., 1., 1.]), 'split4_test_score': array([0., 0., 0., 0., 0., 0.]),
'mean_test_score': array([0.75 , 0.775, 0.75 , 0.75 , 0.725, 0.7  ]),
'std_test_score': array([0.38729833, 0.39051248, 0.38729833, 0.38729833,
0.39051248,
       0.4       ]), 'rank_test_score': array([2, 1, 2, 2, 5, 6], dtype=int32)}
0.775
{'pca__n_components': 1, 'scaler__with_std': False}
```

In order to create a model with the optimal parameters we need to use `gridsearch.best_params_`, which holds a dictionary of each parameter and its optimal value. To 'use' these values we need to update the parameters held in our returned pipeline, `pipe`, by using the `.set_params` method. We use `**` to unpack the dictionary values, tutorial on unpacking here.

Notice that the `gridsearch` has a few attributes of interest. This includes all the results of the kfold rounds `.cv_results_`, the best score obtained `.best_score_`, and the ideal set of parameters to use in the pipeline `.best_params_`. This lase one `.best_params_` will be use to update our `pipe` pipeline for prediction.

By Michael Mann, Steven Chao, Jordan Graesser, Nina Feldman

Supported by:

GW | Geography