# Rasterio Tutorial

***David Gerstenfeld, Adrian Terech***

***November 18, 2024***

### 1.0 Introduction

This tutorial will showcase tools from the Rasterio library for Python. To showcase the basics of Rasterio, we are doing a mock analysis for the City of Philadelphia Planning Department and Sustainability Office on Urban Heat Island Effect. This study will be done by comparing the current land cover & tree canopy rates across the city. Rasterio is a Python library designed for reading and writing geospatial raster data. It provides a high-level API to interact with raster datasets, particularly those stored in formats like GeoTIFF. Raster data typically represents satellite imagery, aerial photography, or any spatially continuous variable (e.g., elevation or temperature) as a grid of pixels or cells.

The Rasterio library handles reading and writing GeoTIFFs and other associated forms of geographic metadata. It integrates well with the Geospatial Data Abstraction Library (GDAL), allowing access to spatial reference systems, projections, and other geospatial metadata. It enables easy reading of specific windows or blocks of large raster datasets without loading the entire file into memory. The raster data can be loaded directly into NumPy arrays for efficient numerical operations. Rasterio also allows reading and transforming coordinate reference systems, making it easy to project raster data into different spatial reference systems.

Key features of Rasterio include:

- Reading and Writing GeoTIFFs: It can handle a variety of raster data formats (GeoTIFF, JPEG2000, etc.) with geographic metadata.
- Geospatial Metadata Handling: Rasterio integrates well with the Geospatial Data Abstraction Library (GDAL), allowing access to spatial reference systems, projections, and other geospatial metadata.
- Data Access: It enables easy reading of specific windows or blocks of large raster datasets without loading the entire file into memory.
- NumPy Integration: The raster data can be loaded directly into NumPy arrays for efficient numerical operations.
- Coordinate Reference Systems: Rasterio allows reading and transforming coordinate systems, making it easy to project raster data into different spatial reference systems.

In this tutorial we will cover reprojection, masking by using polygons, reclassifying rasters, zonal statistics, color coding, and using matplotlib to prepare a final map for the output raster. Our tutorial will use Rasterio and Geopandas to process the data, NumPy to conduct

statistical analysis, and then Matplotlib for data visualization.

***Datasets Used***

**Shapefile Used**

"PHL_Census_Tracts_2021.shp"

**Raster files used**

NLCD_TreeCoverCanopy_PhiladelphiaRegion_2021.tif

NLCD_LandCover_PhiladelphiaRegion_2021.tif

Land_Surface_Temperature_Lansat_2021.tif

### 1.0.1 Rasterio Installation & Data Preparation

We recommend installing Rasterio using anaconda within the Pysal geospatial library (We recommend that you install Rasterio using the Gus5031 env). To install, open the Miniconda prompt, navigate to the proper environment, and use the following commands:

```
conda create -n gus5031 -c conda-forge pysal geopandas  #Installs Pysal which include Rasterio and Geopandas
conda activate gus5031  #The environment our class is using for tutorials
```

*1.1 Importing all neccessary libraries and specific functions*

```
import pysal
import os
import geopandas as gpd
import numpy as np
import rasterio
import fiona
import rasterio.mask
import matplotlib.pyplot as plt
from rasterio.warp import calculate_default_transform, reproject, Resampling
```

*1.2 Setting Workspace and Labeling of Initial Variables*

Below code sets current directory as the workspace using the os library **getcwd()** function. This is then stored in the workspace variable.

```
workspace = os.getcwd()
census_tracts = "PHL_Census_Tracts_2021.shp"
land_surf_temp = "Land_Surface_Temperature_Landsat_2021.tif"
land_cover = "NLCD_LandCover_PhiladelphiaRegion_2021.tif"
tree_cover = "NLCD_TreeCoverCanopy_PhiladelphiaRegion_2021.tif"
landsat_reprojected = 'LST_2021.tif'
landcover_reprojected = 'LC_2021.tif'
treecover_reprojected = 'TCC_2021.tif'
census_prj = 'census_nad_83.shp'
```

*Script Sections are Out of Order for Easy Explanation and Exercise Purposes*

---

**2.0 [Actual Step #7] Color coding, Scaling, Clipping data and Histogram to check data for null and outliers**

The code below defines the variable output path for the Geotif file.

```
# Define output path
output_path = 'heat_island_color.tif'
```

The code uses rasterio to open the *output_path_zonal* raster file as the local variable *src*. Then *src*'s first band is read and stored in the variable data. Then the meta data from *src* is stored in the variable *meta*.

```
# Open input file
with rasterio.open(output_path_zonal) as src:
    data = src.read(1)
    meta = src.meta
```

5.0 is stored in the *max_value* variable as the upper float limit for scaling the raster values.

```
# Define maximum value for scaling
max_value = 5.0
```

The NumPy function **np.clip**, clips the raster values from the input variable data from 0 to the variable *max_value* (which is 5.0). Any values outside of the range are set to the closest range bound. This is all stored in the *clipped_data* variable. Then the *clipped_data* variable is divided by the *max_value* variable and multiplied by the value 5. Finally the resultant data is converted to unsigned 8-bit positive integers using the NumPy **astype()** function.

```
# Scale and clip data
clipped_data = np.clip(data, 0, max_value)
scaled_data = (clipped_data / max_value * 5).astype(np.uint8)
```

Then the metadata is updated again using the update function with the dtype as rasterio unsigned 8-bit integers. An if conditional statement is then used in the updated **meta function** to delete any 'nodata' data within the *meta* variable.

```
# Update metadata without nodata value
meta.update(dtype=rasterio.uint8)
if 'nodata' in meta:
    del meta['nodata']  # Remove nodata setting from metadata
```

Using the **rasterio open** function the updated *meta* variable data is written into the *output_path* variable and the *scaled_data* pixel data is written into *output_path* by using the write function, with one index stated.

```
# Save output file
with rasterio.open(output_path, 'w', **meta) as dst:
    dst.write(scaled_data, indexes=1)
```

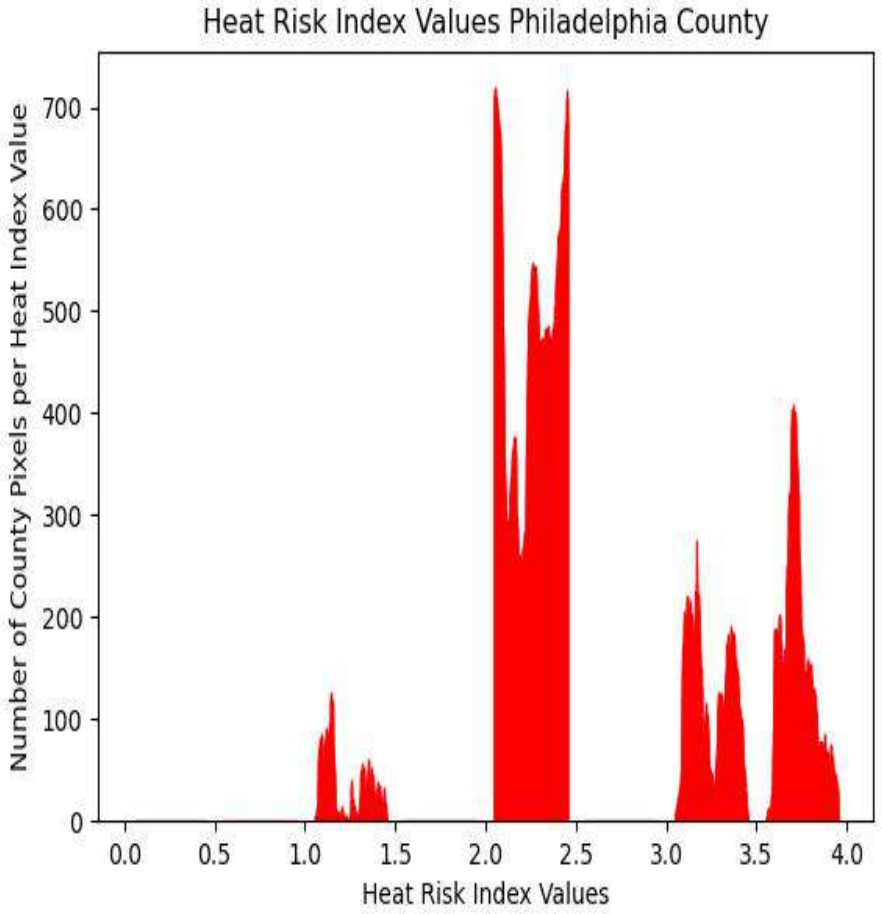
A histogram is made using matplotlib library initially with the **hist()** function. This is using the *scaled_data* input variable, a bins value of 8 for 8 histogram value categories, and edgecolor for color of the histogram shown.

We used a histogram to help confirm that the code had no outliers or null data. Below is the code for the histogram and the output.

```
plt.hist(scaled_data, bins=8, edgecolor='red')
```

The **xlabel, ylabel, and title** functions result in the labeling for the histogram. The **show()** function displays the histogram in a visualization window.

```
plt.xlabel('Num_Of_Instances')
plt.ylabel('Heat_Index')
plt.title('Heat_Index_Philly')
plt.show()
```




As shown by the histogram output of the final processed data, all of the data is higher than 0 and less than 5 (however highest is actually less than 4). There are no null values shown or outliers. This helps confirm the accuracy of the data.

---

**3.0 [Actual Step #5] Reclassifying Rasters**

**Introduction**

In this chapter, we will go over the concept of reclassifying raster datasets and explaining their use for raster analysis. Some common instances where reclassification is used include simplifying datasets with a high number of unique values, reclassifying to only include

values of interest, and assessing land cover change. Throughout this chapter, the reclassification script used in the final product will be split and explained step-by-step.

### 3.1 Preparing to reclassify

For this chapter, we will be observing the reclassification code used for the landsat dataset in the final result. The first section of code opens and acquires information from the specified raster dataset.

```
with rasterio.open("land_surf_temp_mask.tif") as lc:
raster_data = lc.read(1)
profile = lc.profile
```

The 'with' statement is used to open the raster dataset with the use of the **rasterio.open** function, where it is then defined as variable *lc*. The variable *lc* is used in the following two lines to read the first band and metadata of the raster dataset. Variable *raster_data* is used to store the first band within the *lc* dataset, while variable *profile* is used to store its profile metadata. Both variables will be used later in the script and thus, are important to define.

After the variables have been defined, a new NumPy array is created to convert all the existing raster pixel values to 0 using the **np.zeros_like** function as seen below:

```
reclassified_data = np.zeros_like(raster_data)
```

The defined variable *raster_data* from the previous section is added to the function to convert all the raster values within the land cover dataset to 0, giving the user a blank slate to reclassify them in a way that better suits the user.

### 3.2 Reclassification

The next step is to reclassify the dataset with new values. This is accomplished by utilizing a specified number of rules that set newly defined values based on where they fall under the set of rules established by the user.

Here's an example of where a set of rules are used to reclassify the landsat dataset:

```
reclassified_data[(raster_data <= 59.50)] = 1
reclassified_data[(raster_data >= 59.51) & (raster_data <= 69.50)] = 2
reclassified_data[(raster_data >= 69.51) & (raster_data <= 79.50)] = 3
reclassified_data[(raster_data >= 79.51) & (raster_data <= 89.50)] = 4
reclassified_data[(raster_data >= 89.51)] = 5
```

You can also use the **np.round** function to round pixel values before they're reclassified. This can be helpful for reclassifying raster datasets with decimal values. When using the command, it is recommended to include it after the first band of the raster dataset has been read, but also before the NumPy array is created. Here's an instance of where the **np.round** function is used:

```
with rasterio.open("land_surf_temp_mask.tif") as src:
    raster_data = src.read(1)
    rounded_data = np.round(raster_data)
    profile = src.profile

reclassified_data = np.zeros_like(raster_data)
```

In the reclassification section, the variable *raster_data* is substituted by *rounded_data*.

```
reclassified_data[(rounded_data < 60)] = 1
reclassified_data[(rounded_data >= 60) & (rounded_data < 70)] = 2
reclassified_data[(rounded_data >= 70) & (rounded_data < 80)] = 3
reclassified_data[(rounded_data >= 80) & (rounded_data < 90)] = 4
reclassified_data[(rounded_data >= 90)] = 5
```

You can also reclassify raster datasets from a numerical value to a string value in some cases: Example:

```
reclassified_data[(rounded_data < 60)] = ‘Very Low’
reclassified_data[(rounded_data >= 60) & (rounded_data < 70)] = ‘Low’
reclassified_data[(rounded_data >= 70) & (rounded_data < 80)] = ‘Moderate’
reclassified_data[(rounded_data >= 80) & (rounded_data < 90)] = ‘High’
reclassified_data[(rounded_data >= 90)] = ‘Very High’
```

Keep in mind that whenever you're reclassifying a numeric value into a string value, or vice versa, the newly created array needs to specify the new data type used to store the reclassified data. For example:

**Numeric to String**

```
reclassified_data = np.char.array(np.empty_like(raster_data, dtype='<U20'))
```

Note that the **np.zeros_like** function is replaced by the **np.char.array** and **np.empty_like** functions. This is because the **np.zero_like** function is only capable of handling numeric values, while **np.char.array** and **np.empty_like** are capable of handling string values. The data type is also specified in the code to a 20-character limit Unicode array, which is capable of holding string values. The character limit of a Unicode array can be altered to better fit the user, for example 'U50' would establish a 50-character limit on the array.

When reclassifying from a string value to a numeric value, you only need to specify the data type that it is being converted to (either dtype=int or dtype=float). Also note that when you're reclassifying from an integer type to a float value and vice versa, you need to specify the data type.

### 3.3 Saving reclassified rasters

The final step in the reclassification process is to save the reclassified data into a new output file. We'll call it *landsat_reclass* to save the output file to the specified name earlier in the script. Using **rasterio.open**, the output file is opened in a writing mode ('w'), while the metadata is added to ensure that the new output file uses the same metadata as the input file. In the final line, the reclassified data is written into the first band of the output file by using **dest.write**.

```
with rasterio.open(landsat_reclass, 'w', **profile) as dest:
dest.write(reclassified_data, 1)
```

### 3.4 Reclassifications for Final Result

Listed below are the rulesets used to reclassify the land cover and tree cover datasets:

**Land Cover**

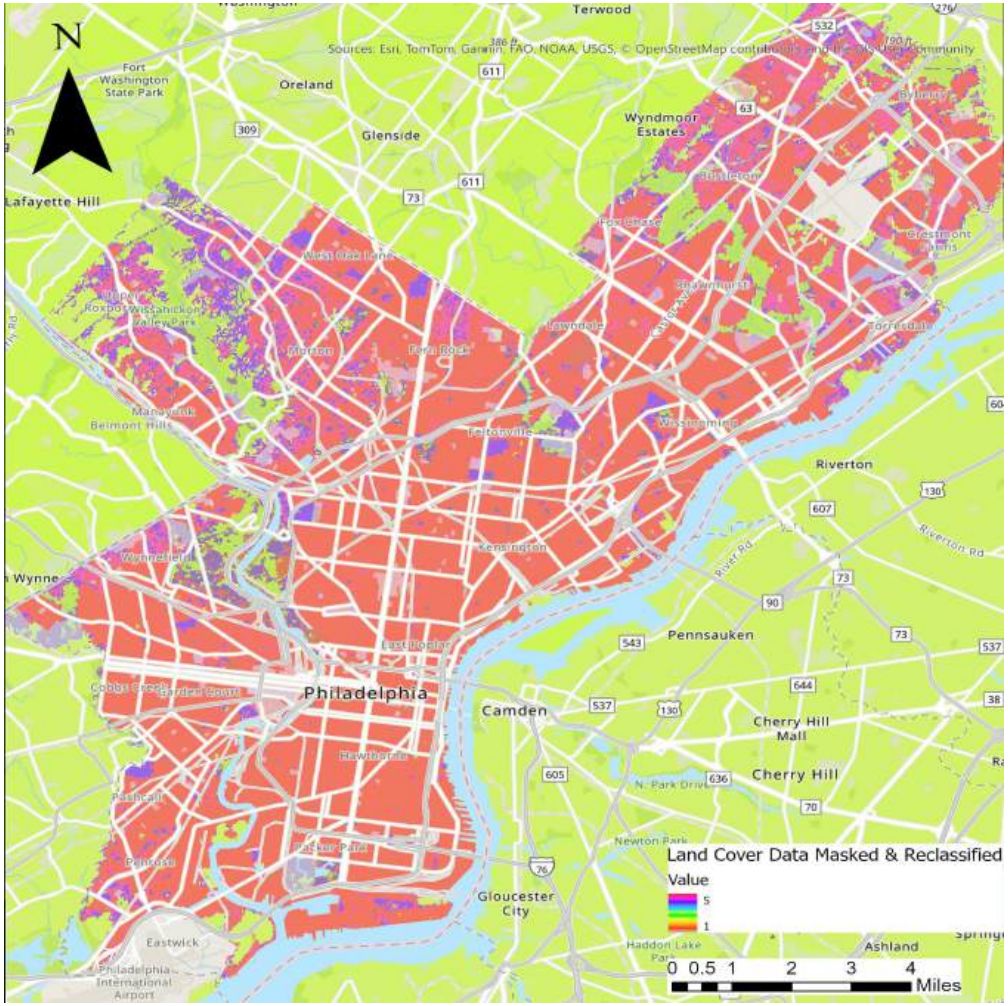
```
reclassified_data[(raster_data > 24) | (raster_data < 21)] = 1
reclassified_data[(raster_data == 21)] = 2
reclassified_data[(raster_data == 22)] = 3
reclassified_data[(raster_data == 23)] = 4
reclassified_data[(raster_data == 24)] = 5
```

**Tree Cover**

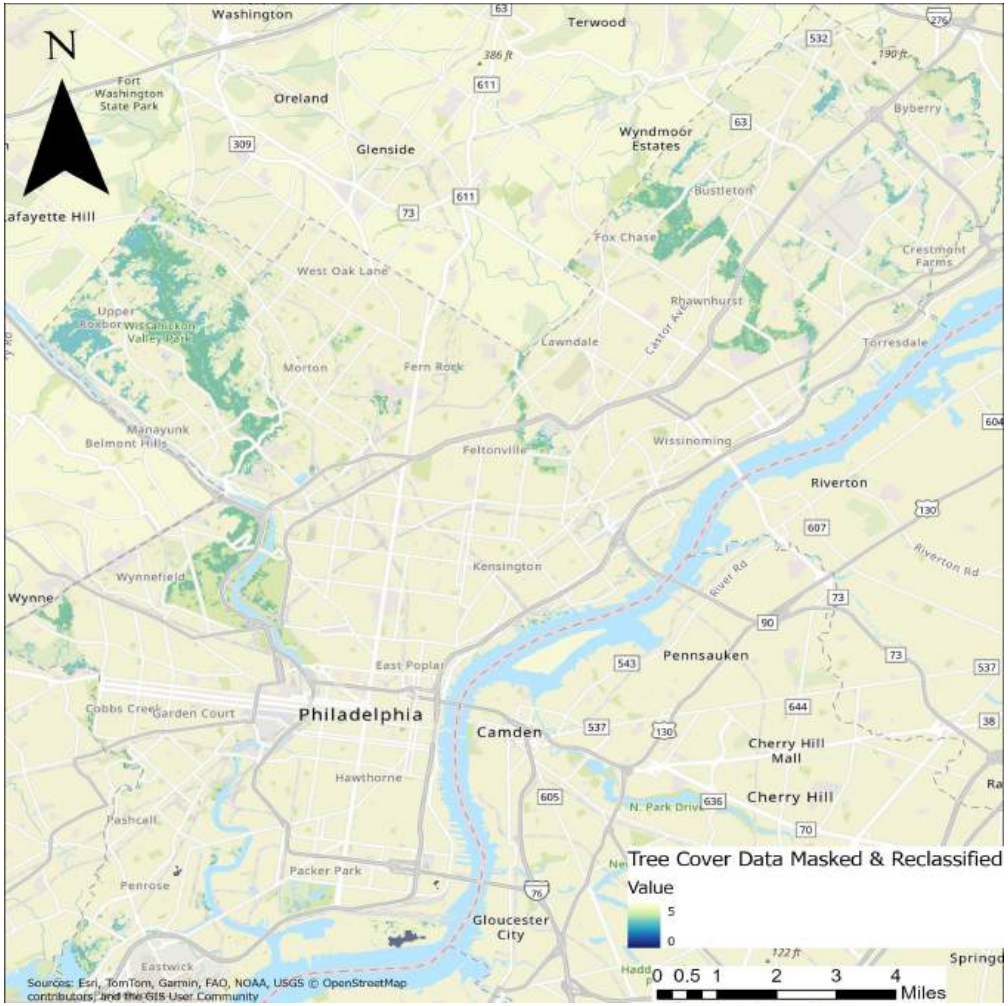
```
reclassified_data[(raster_data >= 0) & (raster_data <= 20)] = 5
reclassified_data[(raster_data >= 21) & (raster_data <= 40)] = 4
reclassified_data[(raster_data >= 41) & (raster_data <= 60)] = 3
reclassified_data[(raster_data >= 61) & (raster_data <= 80)] = 2
reclassified_data[(raster_data >= 81) & (raster_data <= 100)] = 1
```

**Reasonings**

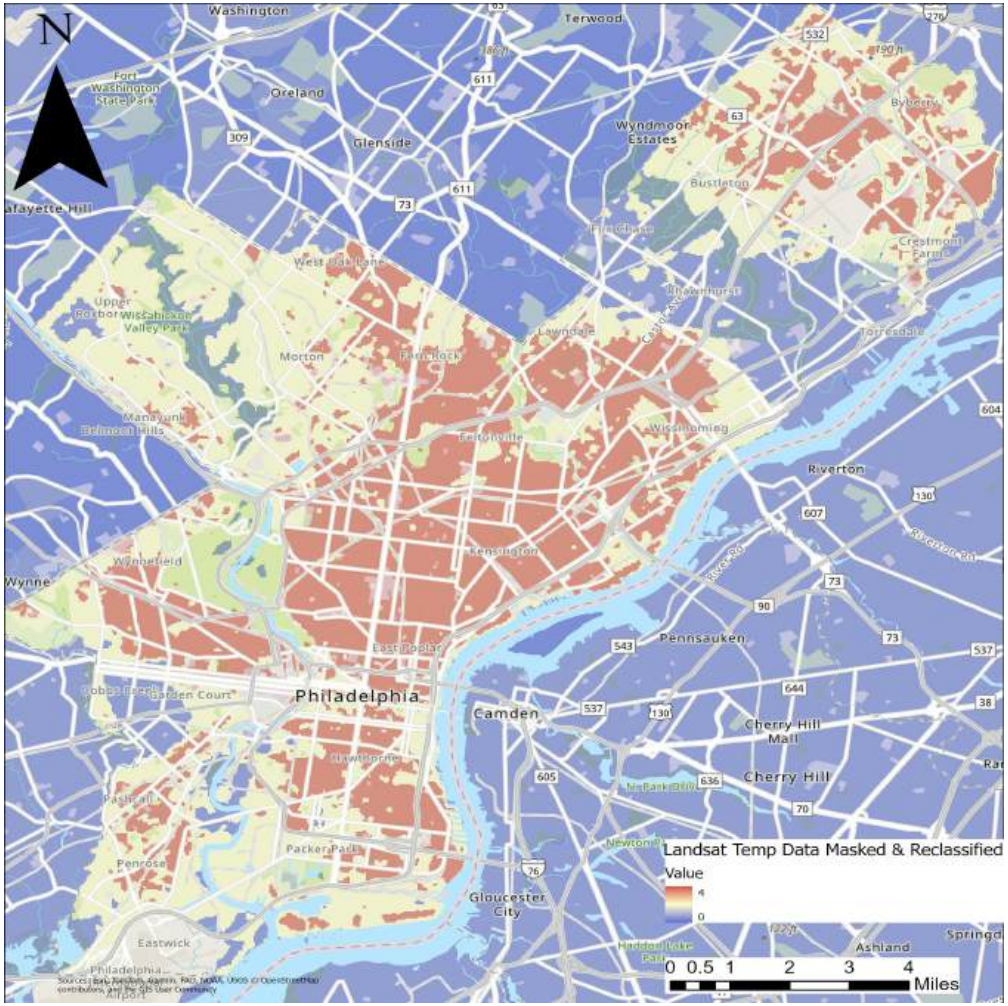
Land Cover was reclassified this way because values 21 to 24 indicate developed land, varying in development intensity (21 is the lowest intenity, 24 is the highest). Values of 1 to 5 were added to reclassified raster, with a high value indicating higher density and higher risk to urban heat island effect.




Tree cover raster was split using the 5-class Jenks (Natural Breaks) method. Since previous journal articles state come to the conclusion that lower tree cover increases risk to urban heat island effect, values were reclassified from 5 to 1.




Landsat data was reclassified into a 5-class method, where the highest and lowest class contain the outlier data while the interior 4 classes are split by 10 degrees. Higher temperature was given a higher reclassified value.




### 3.5 Exercise

*Exercise 1 (Easy):* Lucas County in Northwest Ohio wants to expand its urban forest network to double its current size by 2050. To reach this goal, the city government plans on converting some of the existing open space throughout the city into forest. For this exercise, reclassify water and wetlands classes into 1, forest data (evergreen, deciduous, mixed) into 2, open space developed land into 3, and all other land cover uses into 0. Use the NLCD Land Cover Legend attached below.

https://www.mrlc.gov/data/legends/national-land-cover-database-class-legend-and-description

---

### 4.0 [Actual Step #2] Reprojection of Census Vector Data

The below code uses the **gpd.read_file** function, to read the census_tracts shapefile. The resulting GeoDataFrame is stored in the *gdf* variable. Below that is a print statement which prints the input coordinate system using the command *gdf.crs*.

```
gdf = gpd.read_file(census_tracts)
print("Original CRS:", gdf.crs)
```

The code uses the **gdf.to_crs()** function with the argument input *dst_crs* variable to reproject the gdf variable into the *dst_crs*'s EPSG projection code.

```
gdf_reprojected = gdf.to_crs(dst_crs)
```

The **gdf_reprojected.to_file()** function writes the reprojected GeoDataFrame to a new file with an output file path labeled as the *census_reprojected* variable. The driver 'ESRI Shapefile' stated indicates the format of the output file which is a shapefile.

```
gdf_reprojected.to_file(census_reprojected, driver='ESRI Shapefile')
```

The print statement uses the *gdf_reprojected.crs* command to print the new coordinate reference system EPSG number.

```
print("Reprojected CRS:", gdf_reprojected.crs)
```

---

### 5.0 [Actual Step #3] Reprojecting Raster Data

#### Introduction

In this chapter, we will go over the process of reprojecting raster datasets to a different CRS. This step is essential in optimization and data cleaning to ensure that researchers are getting the most accurate results in their studies. The first part of this chapter will cover how to project a raster dataset to a defined CRS. The second part will cover the use of looping to reproject multiple raster datasets and the use of source rasters as guidelines on reprojection.

### 5.1 Defining CRS and Variables

Before reprojecting, it is essential to determine what CRS you are planning to reprojection a raster dataset to. By using sites like EPSG.io, you can determine the EPSG code for your raster dataset and define it using:

```
dst_crs = 2272 (This is the EPSG code for NAD 1983 State Plane Pennsylvania South, the CRS used for this study
```

The section below opens the raster dataset, in this case the land cover dataset, and defines a wide variety of variables that will be used to reproject the raster dataset. Once the land cover dataset is opened, it is redefined as variable *raster*. The 5 variables defined from the contents of the land cover raster include *src_shape*, *raster_data*, *src_dtype*, *src_transform*, and *src_crs*. Variable *src_shape* is defined by collecting the height and width of the raster dataset. *Raster_data* is defined by reading and collecting the first band, which is then used to determine the variable *src_dtype* by observing the data format used to visualize the first band. *Src_transform* is defined by the affine transformation of the dataset, which <>. And lastly, variable *src_crs* is defined by the land cover raster's existing CRS.

```
with rasterio.open(land_cover) as raster:
    src_shape = (raster.height, raster.width)
    raster_data = raster.read(1)
    src_dtype = raster_data.dtype
    src_transform = raster.transform
    src_crs = raster.crs
```

**5.2 Reprojecting the raster**

Once the necessary variables are defined, new reprojection variables need to be defined under the *dst_crs*. This process is conducted using the **rasterio.warp.calculate_default_transform** function. This function converts a raster dataset to a newly defined CRS, which we have done at the beginning with variable *dst_crs*. The *dst_crs* is added into the calculate_default_transform function alongside variables like *src_crs*, *raster.width*, *raster.height*, and *raster.bounds* to define three new variables.

```
dest_transform, dest_width, dest_height = rasterio.warp.calculate_default_transform(
    source_crs, dst_crs, target_raster.width, target_raster.height, *target_raster.bounds)
```

The three new variables *dest_transform*, *dest_width*, and *dest_height* are all defined by reprojecting the width, height, and bounding box of the raster dataset into the new CRS. After the three variables are defined, an empty destination array named *destination* is created to

contain the height and width of the reprojected raster using the defined data type from the original raster dataset. Listed below is the destination array:

```
destination = np.empty((dest_height, dest_width), dtype=src_dtype)
```

After the destination array is created, the reprojection process can be completed. Using the **reprojection** function, the first band is entered into the function under variable *source*. The recently created destination array is also added into the reprojection under the same name. From both the original raster and reprojected rasters, the Affine transformation and CRS are added into the reprojection function under variables *src_transform*, *src_crs*, *dest_transform*, and *dst_crs*. Lastly, the variable *resampling* is defined by using the *Resampling* function to resample the raster pixel values based on their nearest neighbors. Other resampling methods including bilinear and cubic (using either **Resampling.bilinear** or **Resampling.cubic**) can be substituted to produce more accurate results, but will take longer to process.

```
reproject(
    source=raster_data,
    destination=destination,
    src_transform=src_transform,
    src_crs=src_crs,
    dest_transform=dest_transform,
    dst_crs=dst_crs,
    resampling=Resampling.nearest
)
```

The final step in reprojection is saving the results from the **reproject** function into an output file. Much like in previous steps in the reprojection process, multiple variables are utilized when the output file *landcover_reprojected* is opened. Many of the reprojected variables determined in previous steps are saved into the output file including height, width, CRS, and Transformation. The one exception is the data type as we want to keep the same data type from the original raster for precision. The variable *count* is determined based on how many bands the raster dataset contains. Most raster datasets contain only 1 band, hence in most cases the count will remain 1. But in rare instances where RBG or multi-spectral imaging is used, the variable will need to be changed to either 2 or 3. Lastly, the data will be saved by adding the destination array into the output raster by using *dest.write*. The number in this function once again will be determined based on how many bands the raster contains.

```
with rasterio.open(
    landcover_reprojected,
    'w',
    driver='GTiff',
    height=dest_height,
    width=dest_width,
    count=1,
    dtype=src_dtype,
    crs=dst_crs,
    transform=dest_transform
) as dest:
    dest.write(destination, 1)
```

### 5.3 Reprojection Looping

There are some additional ways to optimize the reprojection process, especially when dealing with multiple rasters. One method is to use looping to reproject multiple rasters at once. To complete a reprojection loop, the original code does need to be altered in a few places. The first change is the creation of two additional variables at the start of the script: *input_raster*, and *output_raster*.

```
input raster = input_file_path1, input_file_path2, ….
output raster = output_file_path1, output_file_path2, ….
```

For organization purposes, it would be recommended to add these two variables after the reprojected CRS has been defined and before the raster dataset is read.

A loop now needs to be constructed just before the raster dataset is opened. This can be accomplished by using a for loop that combines the input_raster and output_raster paths into a pair. For this reason, it is essential to ensure that each of the file paths match with each other (for example: land_cover (input) and land_cover_reprojected (output)).

```
For input_raster, output_raster in zip(input_rasters, output_rasters):
```

The with loop used earlier in the chapter is used below the for loop with minimal changes, with variable *input_raster* being opened and defined in this case.

```
with rasterio.open(input_raster) as raster:
    src_shape = (raster.height, raster.width)
    raster_data = raster.read(1)
    src_dtype = raster_data.dtype
    src_transform = raster.transform
    src_crs = raster.crs
```

Throughout the rest of the reprojection code, the steps remain the same until the end where the reprojected data is saved into an output file. Use variable *output_raster* when opening the output file to successfully complete the loop.

```
with rasterio.open(
    output_raster,
    'w',
    driver='GTiff',
    height=dest_height,
    width=dest_width,
    count=1,
    dtype=src_dtype,
    crs=dst_crs,
    transform=dest_transform
) as dest:
    dest.write(destination, 1)
```

### 5.4 Source Rasters

An additional method that can be used is using source rasters. Source rasters are raster datasets that have already been reprojected and can be used to reproject another raster dataset. The use of source rasters can help speed up the process of reprojection by granting the user the ability to bypass using the **calculate_default_transform** function. To replace this function, both the source raster and the raster that is going to be reprojected will need to be read. Let's suppose that in this case, *land_cover* has already been reprojected. We will use it as a source raster to reproject *tree_cover* to the same projection as *land_cover*. In this section, *land_cover* will be redefined as *src*, while *tree_cover* will be redefined as *target*.

```
with rasterio.open(land_cover) as source:
    source_shape = (source.height, source.width)
    source_data = source.read(1)
    source_dtype = source_data.dtype
    source_transform = source.transform
    source_crs = source.crs

with rasterio.open(tree_cover) as target:
    target_shape = (target.height, target.width)
    target_data = target.read(1)
    target_dtype = target_data.dtype
    target_transform = target.transform
    target_crs = target.crs
```

Since the **calculate_default_transform** function is no longer being used, we can skip that section. The destination array when using a source raster should use the height and width from the source raster, while the target raster's data type should be used. For example:

```
destination = np.empty((source_height, source_width), dtype=target_dtype)
```

The **reprojection** function should also reflect these changes, where the source raster's transformation and CRS should be used as the reprojected data. The target raster's data type, transformation, and CRS should be used as the original raster datasets as shown below:

```
    reproject(
        source=target_data,
        destination=destination,
        target_transform=target_transform,
        target_crs=target_crs,
        reproject_transform=source_transform,
        reproject_crs=source_crs,
        resampling=Resampling.nearest
    )
```

Saving the newly reprojected raster file is similar in this instance, with minor changes made to reflect the variable names used in this section of code. Note that the height and width used for the reprojected raster is from the source raster's height and width defined earlier in the code.

```
with rasterio.open(
   treecover_reprojected,
     'w',
   driver='GTiff',
   height=source_height,
   width=source_width,
   count=1,
   dtype=target_dtype,
   crs=reproject_crs,
   transform=reproject_transform
) as dest:
   dest.write(destination, 1)
```

It is possible to use both reprojection looping and source rasters in the same script. This can be done by combining the steps mentioned above in section 5.3 and 5.4 by defining input and output paths, reading the source raster, and using reprojection looping to conduct the reprojection process. Also note that in this case, the **calculate_default_transform** should be removed from the reprojection loop and the variables be defined to reflect the steps used in section 5.4.

### 5.5 Exercises

*Exercise 1 (Easy):* The City of Philadelphia is planning on conducting a study on how much of the city is covered by car infrastructure. A raster dataset containing impervious surface cover will be used, however the dataset is in a different projection than the city's other datasets. Convert the Impervious Surface Cover dataset into NAD 1983 State Plane Pennsylvania South.

*Exercise 2 (Advanced):* On the other side of the Delaware, Camden is getting ready to conduct a study on Urban Heat Island using Land Cover and Tree Cover datasets. However, both of the rasters are not set to the default reference system used by the city. Using one of the raster datasets already in their directory, write a loop that converts the two raster datasets into WGS 84 UTM / Zone 18N. Use EPSG.io to search for the EPSG code of WGS 84 / UTM Zone 18N.

---

## 6.0 [Actual Step #4] Masking Raster Data Using Polygons from Census Data

### 6.1 Defining and reading geometries

The script below masks three raster files by using a vector shapefile's polygonal outline. The census file below contains an outline for Philadelphia County, the study area for this analysis. The first two lines of code create two lists: *input_files* and *output_files*.

```
# File paths
input_files = [landcover_reprojected, treecover_reprojected, landsat_reprojected]
output_files = ["land_cover_mask.tif", "tree_cover_mask.tif", "land_surface_temp_mask.tif"]

# Read the geometry shapes from shapefile
with fiona.open(census_reprojected, "r") as shapefile:
    shapes = [feature["geometry"] for feature in shapefile]
```

The code uses the Fiona Python library to open the census reprojected shapefile and reads it into a variable named *shapefile*. Then the shapefile variable has the column for geometry called within each feature or row of the shapefile. This data is then stored in the variable *shapes*.

### 6.2 Masking raster datasets

The code then starts a loop by using the zip function to loop through the lists *input_files* and *output_files* simultaneously with the variables named *input_path* and *output_path*. The Rasterio Python library is used to open the *input_path* variable as it is being looped through the *input_files* list. It then stores the *input_path* variable as the *src* variable while under "with". Then the variables *out_image* and *out_transform* are set equal to the result of the **rasterio.mask** function. The *out_image* variable stores the masked raster data as a NumPy array. The *out_transform* variable stores the updated transformation matrix for the clipped raster. The function uses *src* for the input file argument, *shapes* for the geometry data argument, and crop=True for the argument that crops the *src* raster data to the geometry extents of *shapes*, which contains the census vector shapefile. Finally, the *src* meta data for the original input rasters are stored in the new variable *out_meta*.

```
# Loop through each raster, apply mask, and save the output
for input_path, output_path in zip(input_files, output_files):
    with rasterio.open(input_path) as src:
        # Mask the raster with the shapefile geometries
        out_image, out_transform = rasterio.mask.mask(src, shapes, crop=True)
        out_meta = src.meta



   # Update metadata
    out_meta.update({
        "driver": "GTiff",
        "height": out_image.shape[1],
        "width": out_image.shape[2],
        "transform": out_transform
    })
```

### 6.3 Saving output files

The above function updates the *out_meta* variable to reflect the new metadata output of the masking function. The driver makes sure the new file type is a Geotiff, the height argument and width arguments takes its data from the geometry of the new *out_image* variable from the mask function result. The transform argument uses the *out_transform* variable which also uses the result data from the masking function.

```
with rasterio.open(output_path, "w", **out_meta) as dest:
    dest.write(out_image)
```

**Rasterio.open** is used to open and write to the *output_path* and *out_meta* variables data as the variable “dest”. The variable *out_image* is then written or added to the *dest* variable which is the same as the *output_path* variable.

```
print(f'{input_path} has been masked and saved to {output_path}')
```

This print statement prints out the input_path and output_path names every time a loop is completed. For our study, the loop will be completed three times.

### 6.4 Masking with multiple shapefiles

If you want to construct a script where multiple vector shapefiles are used to mask a raster dataset, you are able to do so by adding *shapes* as a variable at the beginning of the script.

```
input_files = [landcover_reprojected, treecover_reprojected, landsat_reprojected]
output_files = ["land_cover_mask.tif", "tree_cover_mask.tif", "land_surface_temp_mask.tif"]
shapes = [census_reprojected, blockgroup_reprojected, block_reprojected]
```

The for looping mentioned in 6.2 would be moved up a few extra lines to include the reading geometry section of the code. Variable *shapes* would also be added into the for loop and **zip()** function. The rest of the script remains the same, with the exception of census_reprojected from the previous code being changed to *shapes* to successfully complete the loop.

```
for input_files, output_files, shapes in zip(input_files, output_files, shapes):
    with fiona.open(shapes, "r") as shapefile:
        shapes = ([feature["geometry"] for feature in shapefile])

    # Apply mask to raster
    with rasterio.open(input_path) as src:
        out_image, out_transform = rasterio.mask.mask(src, shapes, crop=True)
        out_meta = src.meta

        # Update metadata to reflect the new dimensions and transform
        out_meta.update({
            "driver": "GTiff",
            "height": out_image.shape[1],
            "width": out_image.shape[2],
            "transform": out_transform
        })

    # Save the masked raster to the output file
    with rasterio.open(output_path, "w", **out_meta) as dest:
        dest.write(out_image)
```

### 6.5 Exercises

*Exercise 1 (Easy):* A member of a community activist group is awaiting their result for the 2023 copy of the land cover dataset to assess how the census tract has changed in the past year and the impacts it might have to the local nature preserve at the northern end of the census tract. However, the dataset arrived unmasked and includes a larger area than the census tract. Mask the land cover dataset to only include data from within the census tract.

*Exercise 2 (Advanced):* State Representative Margaret Croke and Congressman Mike Quigley have established a joint cooperative to improve tree canopy cover within their respective districts. However, their districts don't entirely overlap with each other and some areas would require finance from a different member of legislature. Mask the tree canopy dataset using both Margaret's and Mike's districts to only include areas that fall under both of their districts. Hint: Look to section 6.4 for guidance.

---

## 7.0 [Actual Step #6] Zonal Statistics on Raster Outputs Using NumPy

```
# Defining input paths
raster_paths = ['land_cover_mask_reclassified.tif', 'tree_cover_mask_reclassified.tif', 'landsat_mask_reclassi
# Creating output file
output_path_zonal = 'heat_island_effect.tif'

# Opening input rasters
with rasterio.open(raster_paths[0]) as src:
  meta = src.meta  # Getting metadata from first raster
  # Reading and stacking all rasters
  stacked_data = np.stack([rasterio.open(path).read(1) for path in raster_paths])

# Calculating the average
average_data = np.nanmean(stacked_data, axis=0)

# Updating metadata
meta.update(dtype=rasterio.float32, count=1, nodata=np.nan)


with rasterio.open(output_path_zonal, 'w', **meta) as dst:
  dst.write(average_data, indexes=1)

print(f"Averaged raster saved as {output_path_zonal}")
```

The script below conducts zonal statistics on three raster files. The first line of code creates a list storing the input raster files within variable *raster_paths*, while the second line of code stores the output file paths under the variable *output_path_zonal*.

```
# Defining input paths
raster_paths = ['land_cover_mask_reclassified.tif', 'tree_cover_mask_reclassified.tif', 'landsat_mask_reclassi
# Creating output file
output_path_zonal = 'heat_island_effect.tif'

# Opening input rasters
with rasterio.open(raster_paths[0]) as src:
  meta = src.meta  # Getting metadata from first raster
  # Reading and stacking all rasters
  stacked_data = np.stack([rasterio.open(path).read(1) for path in raster_paths])
```

Opens the first raster file in the list and stores it in the local *src* variable. The *src* variable meta data is then stored in the *meta* variable. List comprehension (using an iterating variable "path") is then used by the **rasterio open** function to read the first band of each raster in *raster_paths*. This function is used as an argument within the NumPy **np.stack** function to combine all the rasters into a single 3D NumPy array called *stacked_data*. The first dimension corresponds to the number of rasters. The second and third dimensions represent the spatial dimensions which are the rows and columns of the rasters.

```
# Calculating the Piexl-Wise average
average_data = np.nanmean(stacked_data, axis=0)
```

The NumPy nanmean function is then used with an input argument of *stacked_data* and uses the 0 axis which corresponds to the first dimension. This calculates the average for each pixel in the raster array while ignoring all nodata values. The result, *average_data*, is a 2D array representing the averaged raster.

```
# Updating metadata
meta.update(dtype=rasterio.float32, count=1, nodata=np.nan)
```

This code updates the *meta* variable, while confirming the dtype data type is a float32 type raster file. The count sets the number of bands in output raster to 1. Finally, the nodata argument sets any no data value to "NaN".

```
with rasterio.open(output_path_zonal, 'w', **meta) as dst:
  dst.write(average_data, indexes=1)
```

**Rasterio open** function is used on the *output_path_zonal* variable to write the update metadata to it, and write to it as *dst*, the *average_data* stored data with one index.

```
print(f"Averaged raster saved as {output_path_zonal}")
```

This print function outputs the message along with the variable file name for *output_path_zonal*.

---

**8.0 [Actual Step #8] Chloropleth Final Output**

```
plt.imshow(scaled_data, cmap='coolwarm')
plt.axis('off')
cbar = plt.colorbar()
cbar.set_label('Heat Island Risk', labelpad=20)
plt.show()
```

The below line of code uses the matplotlib library **imshow** function to display the input array *scaled_data* as an image. The second argument uses the color map “coolwarm” which represents a continuous range from blue to red which is suited for heat related data visualization.

```
plt.imshow(scaled_data, cmap='coolwarm')
```

The **axis(‘off’)** function removes the x and y axes tick marks as well as labels for a cleaner image.

```
plt.axis('off')
```

The **colorbar()** function adds a colorbar alongside the image to indicate the range of values and their corresponding colors, akin to a legend. It also sets the colorbar equal to the variable cbar.

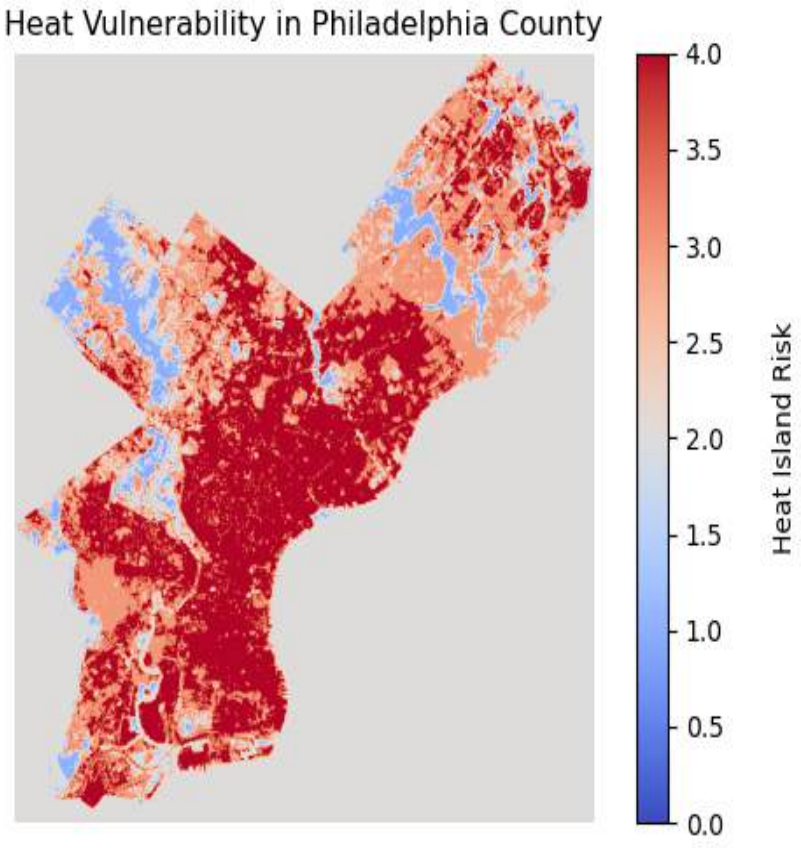
```
cbar = plt.colorbar()
```

The **set_label()** function adds a label to the color bar, and the labelpad argument adds that many units of padding between the color bar and the label.

```
cbar.set_label('Heat Island Risk', labelpad=20)
```

The **show()** function displays the image and color bar in a visualization window.

```
plt.show()
```




***Helpful Links for Resources on Rasterio***

**https://rasterio.readthedocs.io/en/stable/topics/index.html**

**https://geobgu.xyz/py/10-rasterio1.html#**

**DATA SOURCE LINKS**

Land Cover and Tree Canopy Cover: https://www.mrlc.gov/viewer/ Downloaded using custom extent

Landsat Data: https://earthexplorer.usgs.gov/ Downloaded Band 10 dataset and metadata file. Band 10 data came in as raw pixel data, which had to converted to radiance, then to Kelvin, and then to Fahrenheit

Census Tracts: https://www.census.gov/cgi-bin/geo/shapefiles/index.php?year=2021&layergroup=Census+Tracts

Planning Districts: https://opendataphilly.org/datasets/planning-districts/

---

**WORKS CITED**


Amindin, Atiyeh, et al. "Spatial and Temporal Analysis of Urban Heat Island Using Landsat Satellite Images." Environmental Science and Pollution Research, vol. 28, no. 30, 30 Mar. 2021, pp. 41439–41450, https://doi.org/10.1007/s11356-021-13693-0.

Atasoy, Murat. "Assessing the Impacts of Land-Use/Land-Cover Change on the Development of Urban Heat Island Effects." Environment, Development and Sustainability, 29 Nov. 2019, https://doi.org/10.1007/s10668-019-00535-w.

"Documentation — GeoPandas 0.11.0+0.G1977b50.Dirty Documentation." Geopandas.org, geopandas.org/en/stable/docs.html.

Dorman, Michael . "Rasters (Rasterio) — Spatial Data Programming with Python." Geobgu.xyz, 2023, geobgu.xyz/py/10-rasterio1.html. Accessed 16 Nov. 2024.

Matplotlib. "Matplotlib: Python Plotting — Matplotlib 3.3.4 Documentation." Matplotlib.org, matplotlib.org/stable/index.html.

NumPy. "Overview — NumPy V1.19 Manual." Numpy.org, 2022, numpy.org/doc/stable/.

Shahfahad, et al. "Land Use/Land Cover Change and Its Impact on Surface Urban Heat Island and Urban Thermal Comfort in a Metropolitan City." Urban Climate, vol. 41, Jan. 2022, p. 101052, https://doi.org/10.1016/j.uclim.2021.101052.