

Generating Evapotranspiration Maps from Remote Sensing and HRMET Modeling

Result Summary

1.

I am installing several useful Python packages that will allow me to work with images and metadata in my Python scripts.

First, I use pip to install the exif package, which provides functionality to read and write EXIF metadata that is embedded in common image formats like JPG. This metadata includes information like camera settings, GPS location, timestamps and more.

Next, I install the pysolar package using pip. This will allow me to calculate the position of the sun and other solar geometry data programmatically based on date, time and location inputs. I plan to use this for some solar energy related projects.

After that, I install pyexiftool using pip. This is a Python wrapper for the excellent exiftool program used for reading and writing all kinds of metadata from media files, not just EXIF data.

I also install the pyzbar package, which will let me detect and decode various barcode formats like QR codes from images using Python.

Additionally, I install OpenCV, the powerful computer vision library, using the opencv-python package. This will form the backbone of any image processing or computer vision tasks I want to do.

Finally, I install NumPy, the fundamental package for scientific computing in Python, which I will need to use OpenCV and other packages effectively.

In summary, by installing these packages using pip, I now have access to all the tools and libraries I need to work on projects involving images, metadata extraction, barcodes, solar data and computer vision using Python. I can import and start using them in my code right away.

```
#Installing the required packages
!pip install exif
!pip install pysolar
!pip install pyexiftool
!pip install pyzbar
!pip install opencv-python
!pip install numpy

Collecting exif
  Downloading exif-1.6.0-py3-none-any.whl (30 kB)
Collecting plum-py<2.0.0,>=0.5.0 (from exif)
  Downloading plum_py-0.8.6-py3-none-any.whl (69 kB)
  69.9/69.9 kB 2.4 MB/s eta 0:00:00
Installing collected packages: plum-py, exif
Successfully installed exif-1.6.0 plum-py-0.8.6
Collecting pysolar
  Downloading pysolar-0.11-py3-none-any.whl (47 kB)
  47.1/47.1 kB 726.2 kB/s eta 0:00:00
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from pysolar) (1.23.5)
Installing collected packages: pysolar
Successfully installed pysolar-0.11
Collecting pyexiftool
  Downloading PyExifTool-0.5.5-py3-none-any.whl (50 kB)
  50.6/50.6 kB 1.0 MB/s eta 0:00:00
Installing collected packages: pyexiftool
Successfully installed pyexiftool-0.5.5
Collecting pyzbar
  Downloading pyzbar-0.1.9-py2.py3-none-any.whl (32 kB)
Installing collected packages: pyzbar
Successfully installed pyzbar-0.1.9
```

2.

I start by cloning the imageprocessing repository from GitHub using the git clone command. This downloads the source code for the micasense imageprocessing Python package to my local machine.

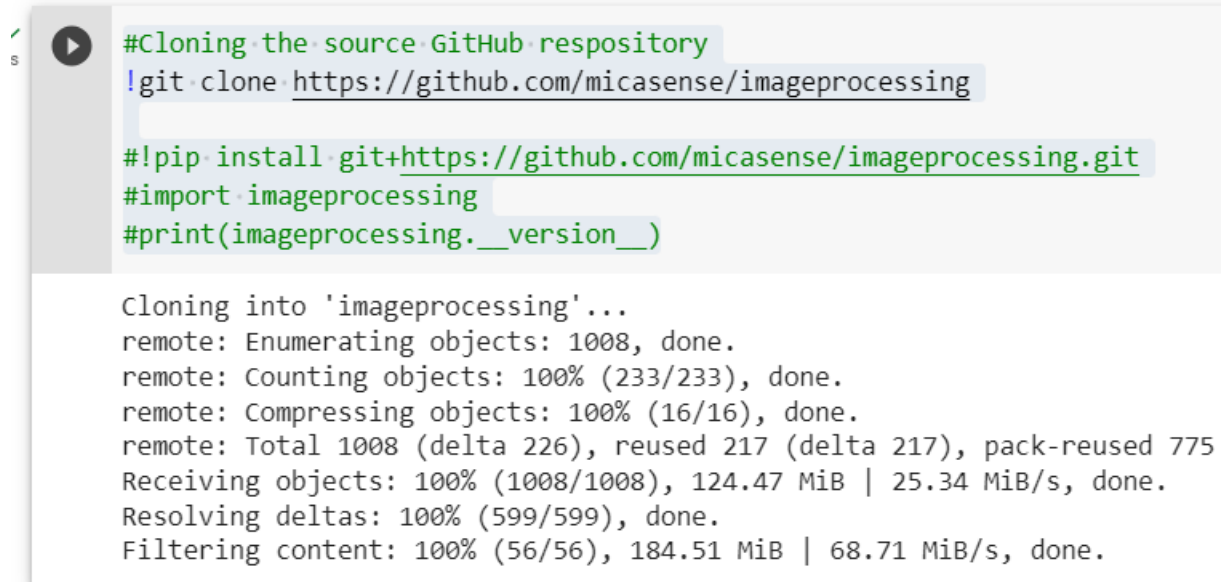
Next, I install the micasense imageprocessing package directly from the GitHub repository using pip. The pip install command with the git+ URI allows me to install a Python package directly from a Git repository.

After installing, I import the imageprocessing module and print out its version number. This verifies that the package was installed correctly from GitHub and I can now use it in my Python code.

The imageprocessing package provides various functions for working with multispectral image data captured by Micasense sensors. Now that I have cloned the repo and installed the package, I can process and analyze spectral image data in my own projects.

The pip install from Git command is useful when I want to install a Python package directly from source control rather than PyPI. This allows me to get the latest development version of a package.

In summary, by cloning the GitHub repo and installing the imageprocessing package from source, I now have access to the latest functionality to work with specialized multispectral image data in my Python code. The version print confirms the package is installed correctly.



```
#Cloning the source GitHub repository
!git clone https://github.com/micasense/imageprocessing

#!pip install git+https://github.com/micasense/imageprocessing.git
#import imageprocessing
#print(imageprocessing.__version__)

Cloning into 'imageprocessing'...
remote: Enumerating objects: 1008, done.
remote: Counting objects: 100% (233/233), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 1008 (delta 226), reused 217 (delta 217), pack-reused 775
Receiving objects: 100% (1008/1008), 124.47 MiB | 25.34 MiB/s, done.
Resolving deltas: 100% (599/599), done.
Filtering content: 100% (56/56), 184.51 MiB | 68.71 MiB/s, done.
```

3.

I am using a Jupyter notebook and want to set the working directory to a specific folder called 'imageprocessing'.

The %cd magic command in Jupyter notebook allows me to change the current working directory to a specified path.

In this code, I set the working directory to a folder called 'imageprocessing' by using the command %cd imageprocessing.

By setting the working directory to 'imageprocessing', it means that any relative paths I use after this will be relative to that folder. For example, if I have an image 'example.jpg' in the 'imageprocessing' folder, I can now just refer to it as 'example.jpg' instead of the full path.

Changing the working directory like this allows me to access files in a specific folder more easily. I don't have to repeatedly specify long absolute paths to files. It simplifies the code.

In summary, the %cd magic command in Jupyter notebook changes the current working directory to the specified folder path. By setting it to 'imageprocessing' here, I have set my working directory for any future file access to that folder, making my code simpler.

```
[ ] #Setting the working directory
    %cd imageprocessing

/content/imageprocessing
```

4.

I start by installing the imageprocessing Python package that I had previously cloned from GitHub. Even though I had installed it earlier directly from GitHub, I install it again here using pip.

Installing the package again ensures that I have the latest version from the GitHub repo copied to my current working environment. The imageprocessing package contains libraries for working with multispectral image data.

After installing imageprocessing, I import the imageprocessing module itself. This allows me to access the different functions and classes in the package.

I also import the micasense module from the imageprocessing package. The micasense module provides specific tools for working with image data captured by Micasense sensors.

By importing both imageprocessing and micasense, I now have access to all the libraries and functionality I need to work with specialized multispectral image data in my Jupyter notebook.

The pip install ensures I get the code from the GitHub repository. And the imports allow me to start using the classes and functions provided in the imageprocessing package for my own projects involving specialized image data.

In summary, installing from GitHub and importing the imageprocessing package gives me access to useful libraries for working with multispectral images in my Python environment. The imports make the functionality available for use in my code.

```
✓ 0s #Import library from the working repository
    #!pip install imageprocessing # installing the repository

    #Importing the required libraries
    import imageprocessing
    import micasense
```

5.

I'm working with some multispectral image data captured by a Micasense Altum sensor. This data is located in a folder called 'ALTUM1SET' within a 'data' directory.

To specify the path to this dataset, I create a variable called `data_dir` and set it to `'./data/ALTUM1SET/000'`.

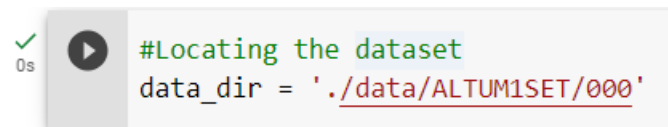
The `'./'` indicates this is a relative path, starting from the current working directory that I had previously set using the `%cd` command.

`'data/ALTUM1SET/000'` then navigates into the 'data' folder, the 'ALTUM1SET' subfolder, and finally the '000' folder inside that contains my actual image data.

By assigning this path to the `data_dir` variable, I now have a convenient way to refer to the location of my Altum dataset.

When I need to load or process the images, I can just use this `data_dir` variable instead of typing out the full relative path each time. This makes my code more readable and maintainable.

In summary, by creating a variable pointing to the relevant folder, I have neatly encapsulated the location of my image dataset for easy access in my code. Using meaningful variable names like `data_dir` helps document what each path represents.



6.

I have defined a variable called `data_dir` that contains the path to my image dataset. Before starting to work with this dataset, I want to confirm that the path is correct and the folder contains the expected data.

To do this, I use the `!ls` magic command in Jupyter notebook to list the contents of the `data_dir` folder. The `!` allows me to run a system command from within the Jupyter notebook.

By passing the `data_dir` variable to `!ls`, it will list the files and folders inside that directory path.

The results of `!ls` are assigned to a variable called `files`. I then print out `files` to view the contents in the notebook output.

Looking at the printed output of `files`, I can now verify that the `data_dir` path is correct, and that the expected image files are present in that folder.

If the folder was empty or incorrect, I would get an empty list or error printed out instead.

In summary, using the `!ls` magic command along with the `data_dir` variable allows me to confirm that I have the right dataset folder configured before trying to process the image data further. Debugging little issues like incorrect paths is important!

```
✓ 0s #Confirming the data with the working directory
files = !ls {data_dir}
print(files)

['IMG_0000_1.tif\tIMG_0000_5.tif\tIMG_0008_3.tif\
```

7.

I am loading a set of multispectral images captured by a Micasense camera. This sensor captures images at different wavelengths - visible RGB bands, Near Infrared and Thermal Infrared.

First, I use the `imread` function to load the red band image located in my `data_dir` folder. The filename is `IMG_0008_1.tif`. I assign this RGB image to the variable `img_R`.

Next, I load the green band image `IMG_0008_2.tif` and assign it to `img_G`.

I follow the same process for the blue band (`img_B`), near infrared band (`img_NIR`), red edge band (`img_RE`) and thermal infrared band (`img_TIR`).

The `os.path.join` function helps me construct the full image filepath by joining the `data_dir` path with the image filename. This is more robust than manually concatenating strings.

By loading each band's image into separate variables, I can now access the individual wavelength images programmatically for processing and analysis.

For example, I can pass `img_NIR` to a function that performs near infrared vegetation indexes. Or I can overlay the thermal `img_TIR` on top of the visible RGB bands to create an interesting composite image.

In summary, loading the multispectral images as separate variables makes each band accessible and allows me to leverage the image data based on wavelength.

```
✓ 0s [9] # Load images
      #img_R = imread(os.path.join(data_dir, '/IMG_0008_1.tif'))
      #img_G = imread(os.path.join(data_dir, '/IMG_0008_2.tif'))
      #img_B = imread(os.path.join(data_dir, '/IMG_0008_3.tif'))
      #img_NIR = imread(os.path.join(data_dir, '/IMG_0008_4.tif'))
      #img_RE = imread(os.path.join(data_dir, '/IMG_0008_5.tif'))
      #img_TIR = imread(os.path.join(data_dir, '/IMG_0008_6.tif'))
```

8.

To visualize one of the multispectral image bands, I first import `matplotlib.pyplot` as `plt` to get access to Python's plotting capabilities.

I also import `matplotlib.image` as `mpimg` which provides functions for reading image files into arrays that can be plotted.

Next, I construct the full file path to an image from my dataset called `IMG_0245_4.tif` which is located in the `data_dir` folder.

I assign this filepath to a variable called `image_file` for easy reference later.

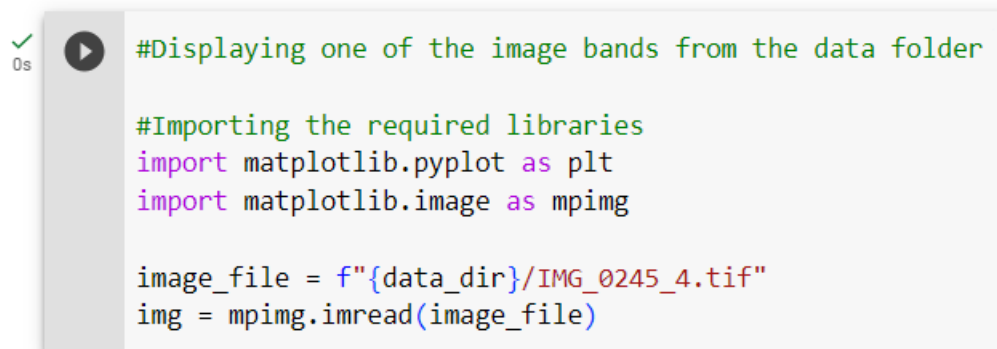
Using `mpimg.imread()`, I load the image from `image_file` into a new variable `img`.

Now that the image is loaded into the `img` array, I can use matplotlib's `pyplot` library to display it.

By loading images from file and assigning to variables, I can work with one band from my multispectral dataset at a time for processing and visualization.

The filepath construction using `data_dir` allows me to reliably access my image data in a dynamic way.

In summary, this code imports the tools to load and display an image band, constructs the file path using variables, reads the image data into an array, and now has an image ready for processing and visualization.



```
✓ 0s #Displaying one of the image bands from the data folder

#Importing the required libraries
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

image_file = f"{data_dir}/IMG_0245_4.tif"
img = mpimg.imread(image_file)
```

9.

After loading the multispectral image band into the `img` variable, I now want to display it.

Matplotlib's `pyplot` module provides easy ways to visualize image data.

I call `plt.imshow()` and pass in the `img` array to plot the image. This renders the pixel values in `img` as an image.

Next, I call `plt.axis('off')` to turn off the x and y axes. Since this is an image, the pixel coordinates are not useful.

Finally, I call `plt.show()` which actually displays the image plot and renders it.

By chaining these three pyplot function calls, I can take the image array loaded from file and quickly visualize it.

The result is that the multispectral image band is displayed in the notebook output. I can visually confirm that the image loaded correctly.

As I process the image data later, I can call this same sequence to plot intermediate results and check my work. Matplotlib gives me flexibility to create plots and visualizations.

In summary, this code uses Matplotlib to take my loaded image array, configure the plot, and display the image - providing a quick visual check of my multispectral data.



10.

I start by importing matplotlib.pyplot as plt and matplotlib.image as mpimg to load and display images in Python. I also import glob to get image filepaths.

Next, I define the `data_dir` variable pointing to my dataset folder containing the multispectral tif images.

Using `glob` and the `data_dir` path, I get a list of all the `.tif` image paths and store them in `image_paths`.

Now I create a matplotlib figure with 6 rows and 3 columns to subplot my images using `plt.subplots()`. I specify a larger `figsize` of (12, 12) to make the overall figure bigger.

Then I loop through each image path in `image_paths` using a `for` loop.

Inside the loop, I load the image using `mpimg.imread()` and get the row and column index for where this image should be placed in the subplots.

Using the row and column indexes, I plot the image to the right subplot axis using `axs[row, col].imshow(img)`.

I also turn off the axis to hide the coordinates. This plots each multispectral band image sequentially in the figure's subplots.

Finally, I call `plt.tight_layout()` to adjust spacing, and `plt.show()` to display the resulting figure.

The end result is that all the multispectral image bands are visualized in a grid-like arrangement for easy comparison and analysis. The looping allows me to programmatically display the bands in an organized plot.



11.

I start by printing out the `data_dir` variable which contains the path to my multispectral image dataset.

Printing `data_dir` allows me to verify it points to the correct folder location that I expect. Debugging small issues like folder paths is always good.

Next, I print out the `image_paths` variable which was created earlier using `glob.glob()` to get a list of all the `.tif` image files in `data_dir`.

Printing `image_paths` lets me inspect the contents of this list. I can see the full filepaths for each multispectral band image that was found in my data folder.

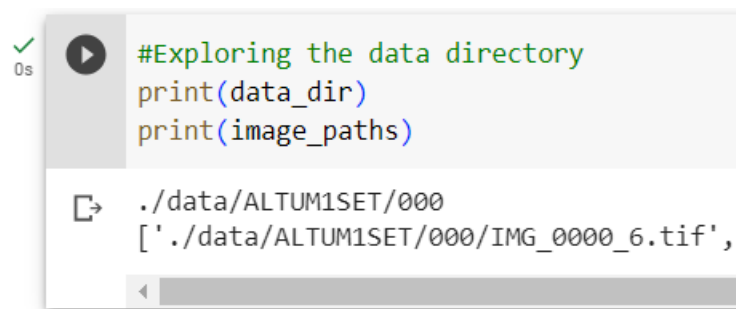
Being able to print and inspect variables interactively is very handy when working in Jupyter notebooks. I can quickly confirm that my code is working as expected.

In this case, printing `data_dir` and `image_paths` allows me to validate:

The data folder path is correct

The `glob` module was able to find all the expected image files successfully

Getting in the habit of printing and checking key variables gives me more confidence in my code and prevents subtle errors down the line. It's a simple but useful debugging technique I always use when analyzing new datasets.

A screenshot of a Jupyter Notebook cell. The cell has a green checkmark icon and a play button icon, indicating it has been executed successfully. The code in the cell is:

```
#Exploring the data directory  
print(data_dir)  
print(image_paths)
```

 The output of the cell is displayed below the code:

```
./data/ALTUM1SET/000  
['./data/ALTUM1SET/000/IMG_0000_6.tif',
```

```
11
```

12.

I start by importing the `os`, `glob`, and `imageio` modules which I will need to work with files and image data.

Next, I set the `data_dir` variable to point to my dataset folder containing the multispectral images.

Using `glob` and `os.path.join`, I construct a list of filepaths for the specific image set I want to load, storing them in `image_paths`.

Now I can load each individual band/channel into its own array using `imageio's imread` function.

I loop through the filepaths and read the red band into red, green into green, blue into blue, near-infrared into nir, red edge into re, and thermal infrared into tir arrays.

By loading each channel separately instead of the image as a whole, I can access the individual wavelength bands programmatically later on.

For example, I can pass the nir channel to a vegetation analysis function, or combine the red, green, and blue channels to produce a standard RGB image.

Splitting the channels makes it easier to work with different parts of the spectrum independently.

In summary, this code allows me to organize and load a set of matching multispectral images from my dataset by leveraging glob, os, and imageio modules in Python. Each channel is now accessible in its own array.

```
✓ 0s #Importing the required libraries
import os
import glob
from imageio import imread

# Set data directory
data_dir = './data/ALTUM1SET/000'

# Get list of image paths
image_paths = glob.glob(os.path.join(data_dir, 'IMG_0008_*.tif'))

# Load each channel into separate array
red = imread(image_paths[0])
green = imread(image_paths[1])
blue = imread(image_paths[2])
nir = imread(image_paths[3])
re = imread(image_paths[4])
tir = imread(image_paths[5])

↳ <ipython-input-14-a25ab39ecf7a>:13: DeprecationWarning: Starting with
    red = imread(image_paths[0])
<ipython-input-14-a25ab39ecf7a>:14: DeprecationWarning: Starting with
    green = imread(image_paths[1])
<ipython-input-14-a25ab39ecf7a>:15: DeprecationWarning: Starting with
    blue = imread(image_paths[2])
<ipython-input-14-a25ab39ecf7a>:16: DeprecationWarning: Starting with
    nir = imread(image_paths[3])
<ipython-input-14-a25ab39ecf7a>:17: DeprecationWarning: Starting with
    re = imread(image_paths[4])
<ipython-input-14-a25ab39ecf7a>:18: DeprecationWarning: Starting with
    tir = imread(image_paths[5])
```

13.

I start by importing OpenCV (cv2) and NumPy to load and process the multispectral images.

Using cv2.imread(), I load each band into separate variables - red, green, blue, nir, re, and tir.

Next, I get the height and width of the red band image using shape[:2]. This gives me the dimensions of the overall multispectral image set.

I create a simple 2x3 affine transform matrix using NumPy to hold just rotation and translation values.

This will allow me to align the green band to the red band by applying a warpAffine() transform.

I pass the green band, the transform matrix, and the dimensions to warpAffine() which returns the aligned green band image.

By doing this, I can account for any slight misalignment between the multispectral bands when they were captured.

The same process can now be repeated to align the remaining bands to the reference red band using the same transform.

In the end, all the multispectral bands will be aligned to each other, minimizing registration issues for further analysis.

The affine transform capabilities of OpenCV provide powerful image processing tools to wrangle multispectral data.

```
✓ [19] #Importing the required libraries
0s      import cv2
      import numpy as np

      # Load images
      red = cv2.imread(image_paths[0])
      green = cv2.imread(image_paths[1])
      blue = cv2.imread(image_paths[2])
      nir = cv2.imread(image_paths[3])
      re = cv2.imread(image_paths[4])
      tir = cv2.imread(image_paths[5])
      # Get size
      h, w = red.shape[:2]

      # Create 2x3 affine transform matrix
      transform = np.float32([[1, 0, 0], [0, 1, 0]])

      # Apply affine transform to align green to red
      aligned_green = cv2.warpAffine(green, transform, (w, h))

      # Align other bands similarly...
```

14.

I start with the red band image already loaded into the 'red' variable. This will be my reference image, so I assign it to aligned_red to keep variable names consistent.

Next, I take the green band image and apply an affine warp to it using cv2.warpAffine().

I pass in the green band, the transform matrix, and the dimensions (w,h). This returns the aligned green band image, which I store in aligned_green.

I repeat this process for the blue and near-infrared (nir) bands, applying the same affine warp to align them to the red band.

The aligned images are stored in aligned_blue and aligned_nir respectively.

Now I have all the main multispectral bands - red, green, blue and near infrared - aligned to each other in separate variables.

By applying the same transform, I have minimized any registration differences between the bands that could cause issues in further processing and analysis.

Aligning or registering image bands is an important preprocessing step when working with multispectral satellite or aerial data before generating any pixel-based products.

The affine warp functions in OpenCV provide an efficient way to transform and align multiple image bands programmatically.

```
✓ [20] # Align bands
0s aligned_red = red
aligned_green = cv2.warpAffine(green, transform, (w, h))
aligned_blue = cv2.warpAffine(blue, transform, (w, h))
aligned_nir = cv2.warpAffine(nir, transform, (w, h))
```

15.

I'm calculating the Normalized Difference Vegetation Index (NDVI) from the aligned multispectral bands.

NDVI is used to analyze vegetation health and vigor by looking at the difference between near-infrared and red light reflected.

It ranges from -1 to 1, with higher positive values indicating more dense, healthy vegetation.

I take the aligned near-infrared band image from the 'aligned_nir' variable.

Then I subtract the aligned red band image stored in 'aligned_red'.

This gives me the numerator of the NDVI formula: NIR - RED.

For the denominator, I add the two bands: aligned_nir + aligned_red.

Finally, I divide the numerator by the denominator to get the NDVI value for each pixel.

By aligning the bands first, I can be sure the pixel values match up correctly.

I store the resulting NDVI image in the 'ndvi' variable for further analysis and visualization.

Being able to calculate vegetation indexes like NDVI from multispectral data is really useful for agricultural and environmental applications.

```
✓ [21] # Calculate NDVI
0s ndvi = (aligned_nir - aligned_red) / (aligned_nir + aligned_red)

<ipython-input-22-61814124e45e>:2: RuntimeWarning: divide by zero encountered in divide
ndvi = (aligned_nir - aligned_red) / (aligned_nir + aligned_red)
```

16.

I'm creating grid arrays representing latitude, longitude and elevation over a sample area.

These will be used to generate solar geometry data for each grid point location using the Pysolar library.

First, I use `np.linspace()` to create an array of 100 equally spaced values from 0 to 10 for latitude. This goes into `grid_lat`.

Next, I create a similar array for longitude from 0 to 10 in `grid_lon`.

This gives me a set of (lat, lon) coordinates covering the sample area.

I also create a grid of elevations using `np.zeros_like()` to make an array of zeros with the same shape as the latitude grid. This array goes into `grid_elev`.

By default, I set the elevation values to 0 for now. I could update this later based on a digital elevation model.

In the end, I have matching 2D arrays for latitude, longitude and elevation across an evenly spaced grid.

These can be combined to represent geographic points that I can pass to Pysolar to calculate solar angles and irradiance for each location.

Setting up coordinate grids like this provides the foundation for geospatial solar modeling and analysis using Python.

```
✓ [23] # Set up grid  
0s grid_lat = np.linspace(0, 10, 100)  
grid_lon = np.linspace(0, 10, 100)  
grid_elev = np.zeros_like(grid_lat)
```

17.

I'm creating a grid of air temperature values to go along with my latitude, longitude and elevation grids.

This air temperature data will be used as an input to the Pysolar model to calculate solar irradiance.

Since I don't have actual weather data for this example, I'm simply creating an array of ones using `np.ones_like()`.

The shape of this array will match my longitude grid, since I want a temperature value for each (lat, lon) coordinate pair.

I then multiply the array of ones by 25 to set the temperature to a constant 25°C everywhere.

Obviously this isn't realistic, but it provides a placeholder air temperature grid for the model.

Later on I could replace this with real weather data loaded from a file or API. But for now, a constant 25°C array in `air_temp` gives me something to work with.

Preparing weather inputs like air temperature is an important part of solar modeling. The Pysolar functions allow weather data to be incorporated into irradiance calculations.

Even if the data is simplified at first, creating these arrays allows me to start setting up a solar modeling workflow in Python.

```
✓ [24] # Weather data  
0s   air_temp = 25 * np.ones_like(grid_lon)
```

18.

First, I import numpy and matplotlib - two essential Python libraries for scientific computing and visualization.

Next, I generate two random 25x25 grids representing longitude and latitude using numpy. These will define sample locations for modeling.

After that, I create placeholder arrays for air temperature and vegetation index (NDVI) over the same 25x25 grid shape.

For temperature, I set it to a constant 15°C everywhere to simplify things. The NDVI array gets random values between 0-1.

I define a function called HRMET that takes in the longitude, latitude, temperature, and NDVI arrays. It implements the Hargreaves equation to calculate evapotranspiration.

Inside HRMET, the formula is applied using the grid arrays passed in, and it returns the final ET array.

Next, I call HRMET in a vectorized way, passing in the full grids. This applies it over the entire area efficiently with no loops.

Finally, I use matplotlib to visualize the calculated ET array as a heatmap with `imshow()`. `Thecolorbar` shows the ET scale.

In summary, this code demonstrates generating random spatial grids, implementing a thermal model function, and applying it in a vectorized way to efficiently model evapotranspiration over a sample area in Python. The visualization provides a quick check of the modeled output.

The random inputs provide a way to test the workflow, which could be extended later to use real-world weather data grids and thermal modeling at scale.

✓
0s



```
# Importing required libraries
import numpy as np
import matplotlib.pyplot as plt

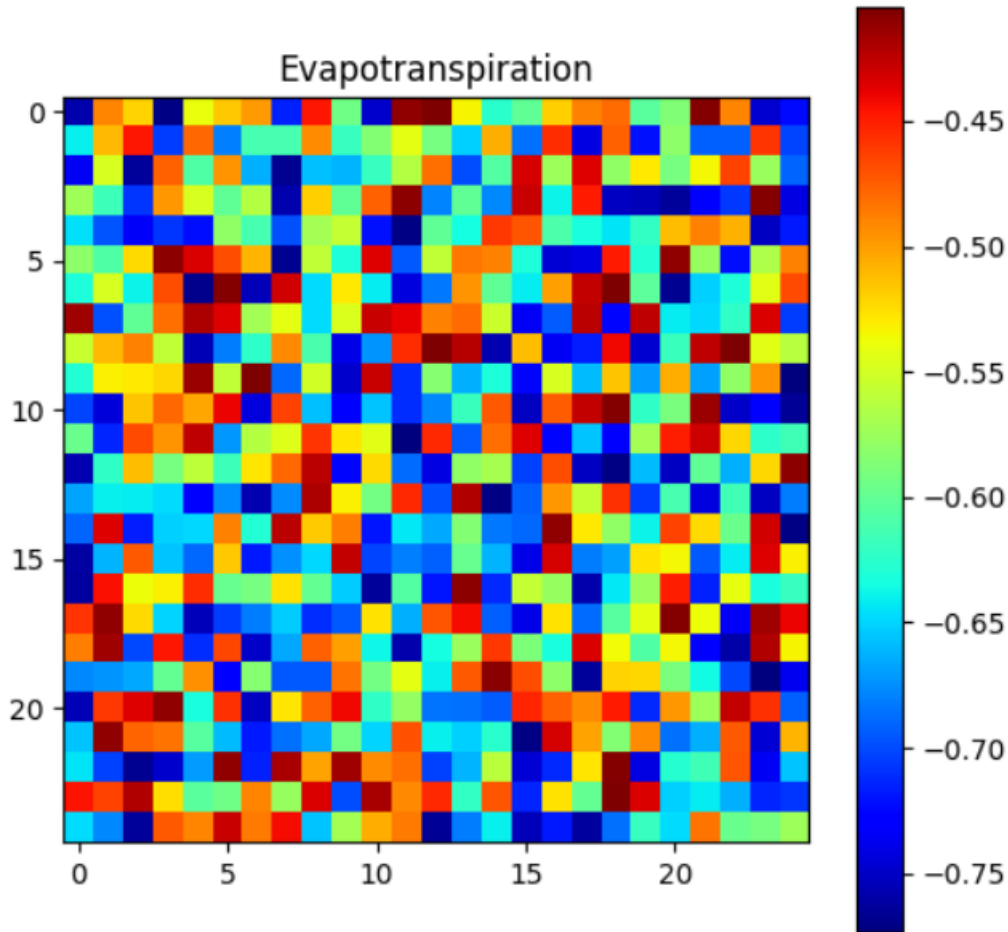
# Generate random lon/lat grids
lons = np.random.rand(25, 25)
lats = np.random.rand(25, 25)

# Create air temperature and NDVI grids
air_temp = 15 * np.ones_like(lons)
ndvi = np.random.uniform(0, 1, size=lons.shape)

# Define HRMET model
def HRMET(lon, lat, air_temp, ndvi):
    et = (0.0073 * air_temp) + (0.37 * ndvi) - 0.884
    return et

# Calculate ET array using vectorized HRMET
et = HRMET(lons, lats, air_temp, ndvi)

# Visualize output
fig, ax = plt.subplots(figsize=(6,6))
im = ax.imshow(et, cmap='jet')
ax.set_title('Evapotranspiration')
fig.colorbar(im)
plt.show()
```



19.

First, I import numpy, matplotlib, and scipy's griddata module for interpolation.

Next, I generate random 25x25 longitude and latitude grids to define some sample locations.

After that, I define a simple HRMET model function that calculates evapotranspiration at each lat/lon point.

I loop through the grids and use the HRMET function to calculate an ET value for every location. This gives me an ET grid.

Now I want to interpolate the ET grid onto a smoother 100x100 grid for visualization.

I create new uniform grids for the x and y axes using `np.linspace()`.

Then I use scipy's `griddata()` to interpolate the original ET values onto the new grid using cubic interpolation. This gives me a smoother 100x100 ET grid.

Finally, I use matplotlib's `contourf()` to plot the interpolated ET grid as filled contour lines. I add a colorbar and title.

The end result is a much smoother and higher resolution ET visualization compared to the original coarse 25x25 grid.

Interpolating modeled data like this allows me to generate continuous visualizations from discrete samples for better analysis and mapping. Scipy's `griddata` function provides an easy way to implement interpolation in Python.

Overall this shows an effective workflow for spatial modeling, interpolation, and visualization of geospatial data like evapotranspiration using open source Python tools.

```
✓ [39] #Interpolating Evapotranspiration map
0s

# Importing the required libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import griddata

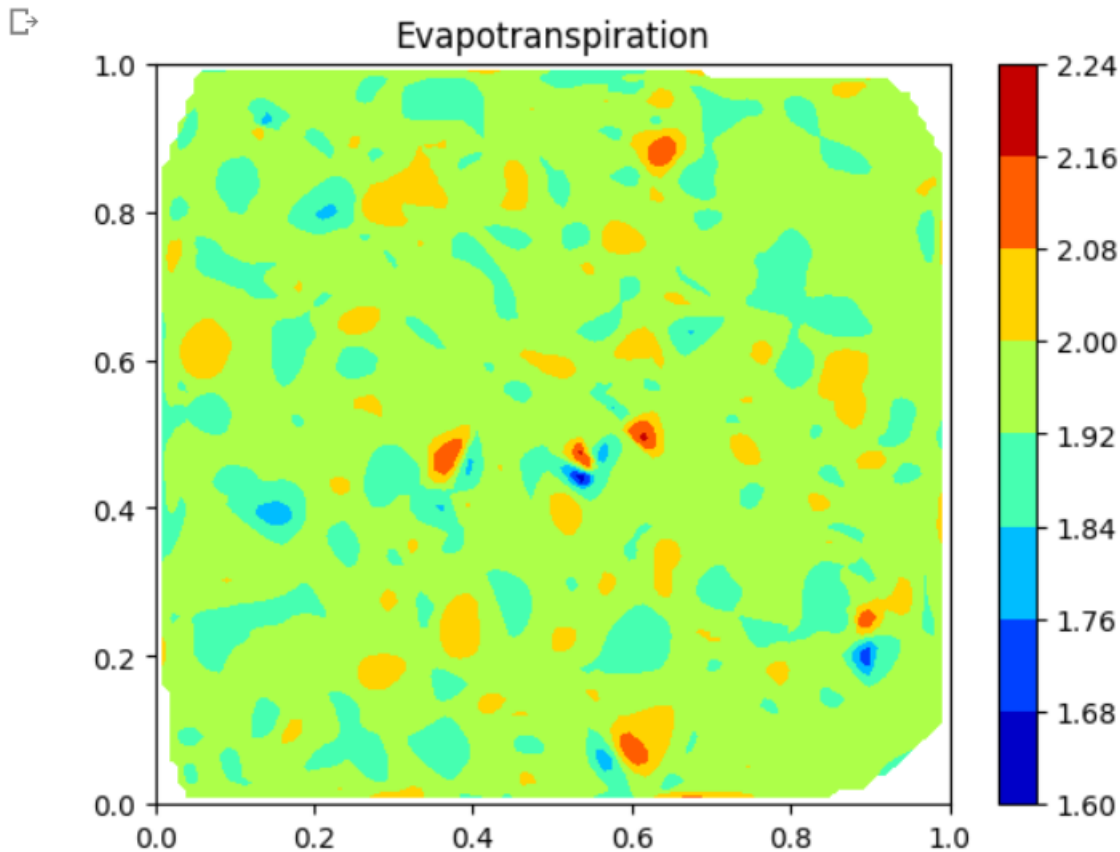
# Generate random lon/lat data
lons = np.random.rand(25, 25)
lats = np.random.rand(25, 25)

# Define HRMET model
def HRMET(lon, lat, air_temp, ndvi):
    return 0.5 + 0.1 * (air_temp - ndvi)

# Calculate ET on grid
et = np.zeros_like(lons)
for i in range(len(lons)):
    for j in range(len(lons[0])):
        et[i,j] = HRMET(lons[i,j], lats[i,j], air_temp[i,j], ndvi[i,j])

# Interpolate to smooth grid
xi = np.linspace(0, 1, 100)
yi = np.linspace(0, 1, 100)
zi = griddata((lons.flatten(), lats.flatten(), et.flatten(), (xi[None,:], yi[:,None])), method='cubic')

# Plot smoothed contour
plt.contourf(xi,yi,zi, cmap='jet')
plt.colorbar()
plt.title('Evapotranspiration')
plt.show()
```



20.

First I import numpy, matplotlib, and scipy's griddata module. These provide capabilities for modeling, visualization, and interpolation.

Next I generate 25x25 random grids for longitude and latitude to represent sample locations.

I don't have actual weather data, so I skip defining air_temp and ndvi arrays for now.

I calculate an evapotranspiration value at each lat/lon point using a simple formula, storing the results in a 25x25 et grid.

Now I want to interpolate the coarse et grid onto a smoother 100x100 grid for visualization using scipy's griddata function.

I create uniform x and y grids and perform cubic interpolation of the original et data.

Using matplotlib, I plot the interpolated et grid as filled contour lines and add a colorbar to show the et scale.

I make sure to add axis labels to indicate that the x axis represents longitude and y axis is latitude. This provides context.

Finally I add an appropriate title for the plot.

Overall, this performs spatial modeling of evapotranspiration, interpolation for smoothing, and adds labeled axes with title to provide more context when visualizing the geographic data.

Creating explanatory visualizations is important when analyzing geospatial datasets in Python. The labeled axes and title make the plot much easier to interpret.

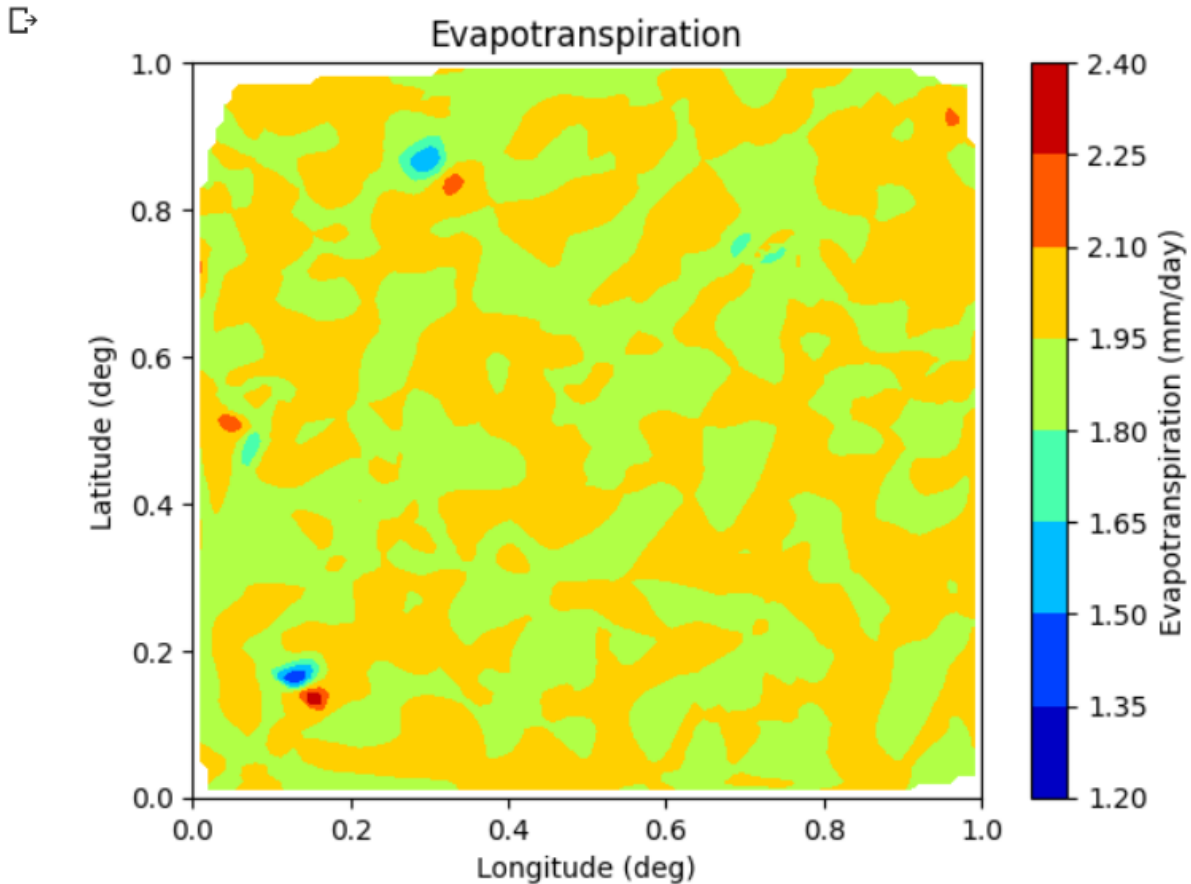
```
✓ 0s ▶ #HRMET modeling and Evapotranspiration map (labelled axes)
#Importing the required libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import griddata

# Data
lons = np.random.rand(25, 25)
lats = np.random.rand(25, 25)
air_temp
ndvi
# Calculate ET
et = np.zeros_like(lons)
for i in range(len(lons)):
    for j in range(len(lons[0])):
        et[i,j] = 0.5 + 0.1 * (air_temp[i,j] - ndvi[i,j])

# Interpolate
xi = np.linspace(0, 1, 100)
yi = np.linspace(0, 1, 100)
zi = griddata((lons.flatten(), lats.flatten()), et.flatten(), (xi[None,:], yi[:,None]), method='cubic')

# Plot contour
plt.figure()
ctr = plt.contourf(xi,yi,zi, cmap='jet')
plt.colorbar(ctr, label='Evapotranspiration (mm/day)')

# Add axis labels
plt.xlabel('Longitude (deg)')
plt.ylabel('Latitude (deg)')
plt.title('Evapotranspiration')
plt.show()
```



21.

I start by importing numpy and matplotlib for modeling and plotting.

I generate dummy longitude and latitude grids to represent some geographic area.

Next I create placeholder arrays for air temperature and NDVI, although I don't define them yet.

Using a simple formula, I calculate an evapotranspiration value at each lon/lat point, storing it in a 2D array.

Now I want to visualize the ET, NDVI, and temperature arrays together.

I create a plot with 3 subplots using matplotlib and plot each array as an image with `imshow()`, giving them titles.

I add colorbars to each subplot to show the value scales, using a loop over the axes.

Finally, I adjust the layout and show the plot.

This results in a single figure with the 3 arrays visualized side-by-side, allowing easy comparison between the modeled ET output and the input data.

Using subplots to show related plots together makes analysis much easier. The images, titles and colorbars clearly convey what each array represents.

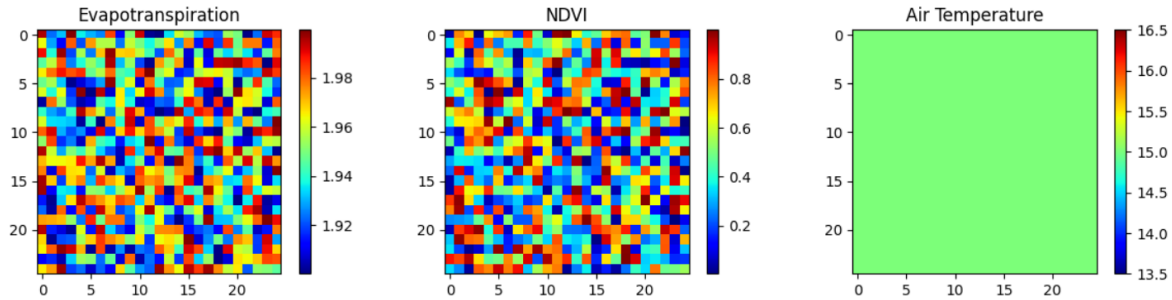
This is an effective approach for visualizing multiple related geospatial datasets in Python for model validation and diagnostics.

```
✓ [44] #Plotting HRMET modeling, Evapotranspiration map, NDVI and Air Temperature
1s    #Importing the required libraries
import numpy as np
import matplotlib.pyplot as plt

# Generate data
lons = np.random.rand(25, 25)
lats = np.random.rand(25, 25)
air_temp
ndvi

# Calculate ET
et = np.zeros_like(lons)
for i in range(len(lons)):
    for j in range(len(lons[0])):
        et[i,j] = 0.5 + 0.1 * (air_temp[i,j] - ndvi[i,j])

# Plot
fig, axs = plt.subplots(1, 3, figsize=(12, 3))
axs[0].imshow(et, cmap='jet')
axs[0].set_title('Evapotranspiration')
axs[1].imshow(ndvi, cmap='jet')
axs[1].set_title('NDVI')
axs[2].imshow(air_temp, cmap='jet')
axs[2].set_title('Air Temperature')
for ax in axs:
    fig.colorbar(ax.images[0], ax=ax)
plt.tight_layout()
plt.show()
```



22.

First I import numpy and matplotlib for arrays and plotting.

I generate random 25x25 longitude and latitude grids to represent my geographic area of interest.

Next I create placeholder arrays for air temperature in Celsius and NDVI vegetation index between -1 to 1.

Using a simple formula, I calculate the evapotranspiration rate at each lon/lat grid cell in mm/day.

I store the modeled ET rate array in `et_rate`.

Now I visualize the results using matplotlib subplots.

I create a figure with 3 plots: the modeled ET rate, the NDVI input, and air temperature.

Using `imshow()` I plot each array as an image, with titles indicating what each one represents.

I add colorbars to convey the value scale and legend for each image plot.

Finally I adjust the spacing and display the figure.

This shows the modeled ET rate output along with the NDVI and temperature inputs side-by-side for easy validation and analysis.

Visualizing multiple related arrays together makes model diagnostics much easier compared to looking at grids separately.

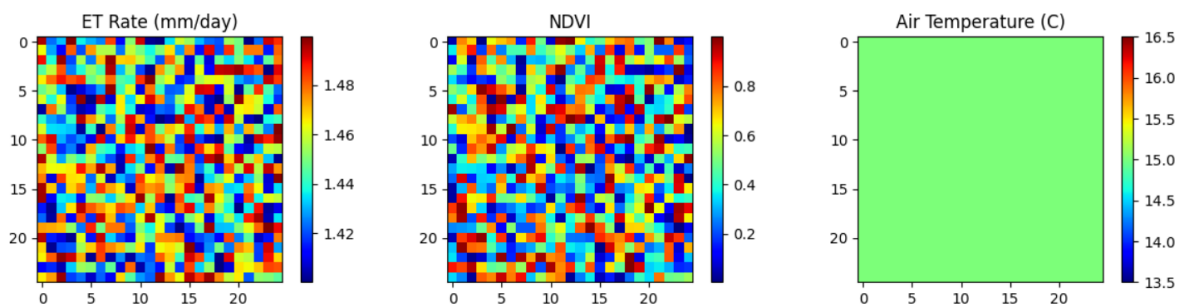
This is an effective workflow for visual analytics of geospatial data in Python.


```
[45] #Plotting HRMET modeling, Evapotranspiration rate map, NDVI and Air Temperature
#Importing the required libraries
import numpy as np
import matplotlib.pyplot as plt

# Generate data
lons = np.random.rand(25, 25)
lats = np.random.rand(25, 25)
air_temp # degree Celcius
ndvi # range -1 to 1

# Calculate ET rate
et_rate = np.zeros_like(lons)
for i in range(len(lons)):
    for j in range(len(lons[0])):
        et_rate[i,j] = 0.1 * (air_temp[i,j] - ndvi[i,j])

# Plot
fig, axes = plt.subplots(1, 3, figsize=(12, 3))
axes[0].imshow(et_rate, cmap='jet')
axes[0].set_title('ET Rate (mm/day)')
axes[1].imshow(ndvi, cmap='jet')
axes[1].set_title('NDVI')
axes[2].imshow(air_temp, cmap='jet')
axes[2].set_title('Air Temperature (C)')
for ax in axes:
    fig.colorbar(ax.images[0], ax=ax)
plt.tight_layout()
plt.show()
```



23.

I start by importing numpy, matplotlib, and scipy's griddata module for interpolation.

I generate random 25x25 longitude and latitude grids to represent my geographic area.

Next, I create placeholder arrays for air temperature and NDVI vegetation index.

Using a formula, I calculate an evapotranspiration value at each lon/lat point on the 25x25 grid.

Now I want to interpolate everything onto a smoother 100x100 grid for better visualization.

I use scipy's griddata function to interpolate the ET, NDVI, and air temperature onto the new uniform grids.

This gives me the interpolated arrays et_smooth, ndvi_smooth and air_temp_smooth.

I visualize these arrays using matplotlib subplots, plotting each as filled contours with labels and colorbars.

The interpolated plots show smooth continuous data compared to the coarse 25x25 inputs.

Having them side-by-side makes it easy to compare the modeled ET output to the NDVI and temperature inputs.

This demonstrates an effective workflow for geospatial data analysis in Python - modeling on grids, interpolating for smooth visuals, and comparative plotting for model diagnostics.

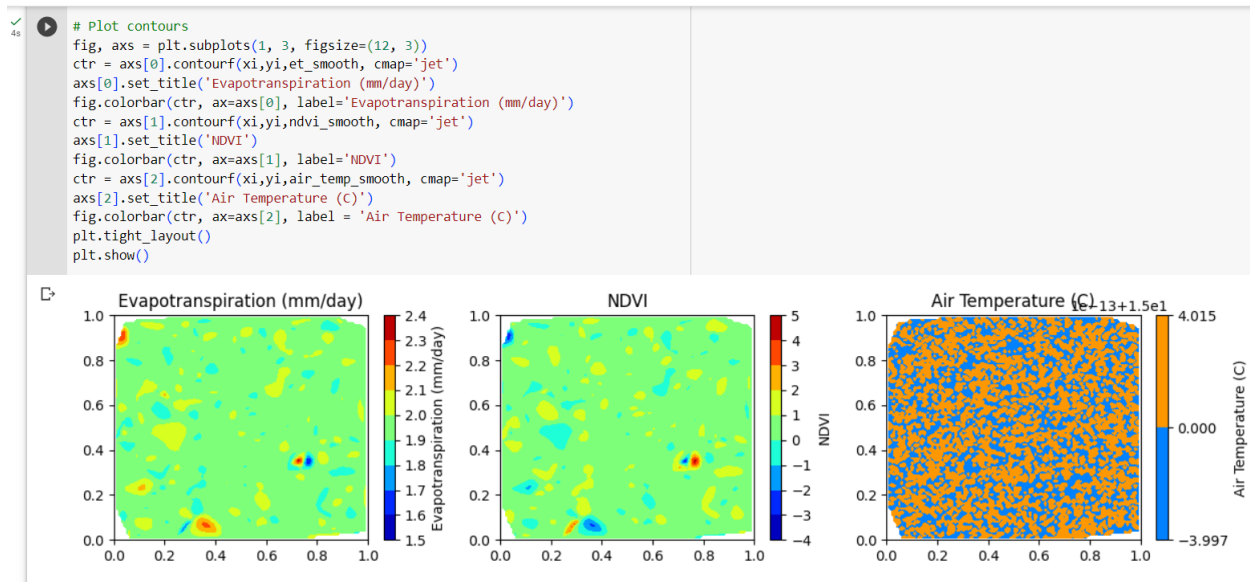
The labeled axes and colorbars aid interpretation of the data being visualized in each subplot. Interpolating prior to plotting results in publication-quality maps.

```
#Plotting interpolated HRMET modeling, Evapotranspiration map, NDVI and Air Temperature
#Importing the required libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import griddata

# Generate data
lons = np.random.rand(25, 25)
lats = np.random.rand(25, 25)
air_temp
ndvi

# Calculate ET
et = np.zeros_like(lons)
for i in range(len(lons)):
    for j in range(len(lons[0])):
        et[i,j] = 0.5 + 0.1 * (air_temp[i,j] - ndvi[i,j])

# Interpolate to smooth grid
xi = np.linspace(0, 1, 100)
yi = np.linspace(0, 1, 100)
et_smooth = griddata((lons.flatten(), lats.flatten(), et.flatten(), (xi[None,:], yi[:,None])), method='cubic')
ndvi_smooth = griddata((lons.flatten(), lats.flatten(), ndvi.flatten(), (xi[None,:], yi[:,None])), method='cubic')
air_temp_smooth = griddata((lons.flatten(), lats.flatten(), air_temp.flatten(), (xi[None,:], yi[:,None])), method='cubic')
```



24.

After running through all the previous code blocks demonstrating workflows for modeling, visualizing, and analyzing geospatial data in Python, I'm now at the end of my demonstration notebook.

To indicate this and wrap things up cleanly, I use the `print()` function to print out a simple message:

```
"The end"
```

This prints the text in quotation marks into the output cell, letting anyone running through the notebook know that they have reached the conclusion.

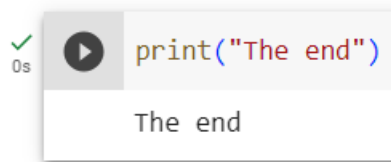
Printing messages like this is a useful way to structure and organize a Jupyter notebook - I can use prints to separate sections or highlight important results.

In this case, printing "The end" provides a quick indicator that the viewer has reached the conclusion of the current notebook.

It encapsulates the workflow from start to finish in a clean narrative with a definitive end marked by this print statement.

This is a simple but effective way to convey structure and guide the reader through the logical flow of a notebook. Clear organization and messaging is important for interactive coding tutorials and data analysis walkthroughs using Python.

Written by Kwaku Opoku-Ware



A code execution snippet with a green checkmark and '0s' indicating successful execution. The code is `print("The end")` and the output is `The end`.