EXERCISE 3
Load topographic derivatives and climate variables

Introduction

You have gained familiarity with Google Earth Engine (GEE), and have used it to create a Landsat composite image and calculate spectral indices for your area of interest. Spectral indices can be predictive of soil classes and characteristics, but there's a breadth of additional spatial data available in the GEE data catalog that we can also use in digital soil mapping. In this exercise, you will continue to explore the GEE data catalog and gain additional practice using native GEE functions, GTAC functions, and writing scripts in order to prepare a suite of spatial variables as inputs to your digital soil map. You will access and calculate topographic derivatives and climate variables to use as additional predictors for your classifications and regressions.

Objectives

- Continue to explore the GEE Data Catalog, accessing the USGS National Elevation Dataset and PRISM climate data
- Use and understand raster functions performed on topographic dataset to calculate topographic derivatives
- Use and understand raster functions performed on climate dataset to calculate climate derivatives
- Practice adapting code into a function to be easily applied across multiple datasets
- Practice reading and interpreting existing code and functions

Required Data:

- **VT_boundary.shp** – shapefile representing example boundary area
- **Pedon data**

Prerequisites

- **Completion of Exercises 1 and 2 (you can review completed code by accessing the course repository)**
- **Google Chrome installed on your machine**
- **An approved Google Earth Engine account**
- **Follow the links below to gain read access to the GEE code respositories we will refer to in the script.**
    - Click here to gain access to the GTAC module repository
    - Click here to gain access to the GTAC training repository

USDA Non-Discrimination Statement

In accordance with Federal civil rights law and U.S. Department of Agriculture (USDA) civil rights regulations and policies, the USDA, its Agencies, offices, and employees, and institutions participating in or administering USDA programs are prohibited from discriminating based on race, color, national origin, religion, sex, gender identity (including gender expression), sexual orientation, disability, age, marital status, family/parental status, income derived from a public assistance program, political beliefs, or reprisal or retaliation for prior civil rights activity, in any program or activity conducted or funded by USDA (not all bases apply to all programs). Remedies and complaint filing deadlines vary by program or incident.

Persons with disabilities who require alternative means of communication for program information (e.g., Braille, large print, audiotape, American Sign Language, etc.) should contact the responsible Agency or USDA's TARGET Center at (202) 720-2600 (voice and TTY) or contact USDA through the Federal Relay Service at (800) 877-8339. Additionally, program information may be made available in languages other than English.

To file a program discrimination complaint, complete the USDA Program Discrimination Complaint Form, AD-3027, found online at How to File a Program Discrimination Complaint and at any USDA office or write a letter addressed to USDA and provide in the letter all of the information requested in the form. To request a copy of the complaint form, call (866) 632-9992. Submit your completed form or letter to USDA by: (1) mail: U.S. Department of Agriculture, Office of the Assistant Secretary for Civil Rights, 1400 Independence Avenue, SW, Washington, D.C. 20250-9410; (2) fax: (202) 690-7442; or (3) email: program.intake@usda.gov.

USDA is an equal opportunity provider, employer, and lender.

Table of Contents

Part 1: Calculate topographic derivatives

Our first step will be to calculate topographic derivatives from the USGS National Elevation Dataset. We have pre-written a script that will calculate the topographic derivatives, which include slope percentage, aspect sine, aspect cosine, elevation.

Like Exercise 2, we have provided the scripts in "worksheet format," with a script outline that contains comments describing what the code will do.

A. Load the exercise script and input data

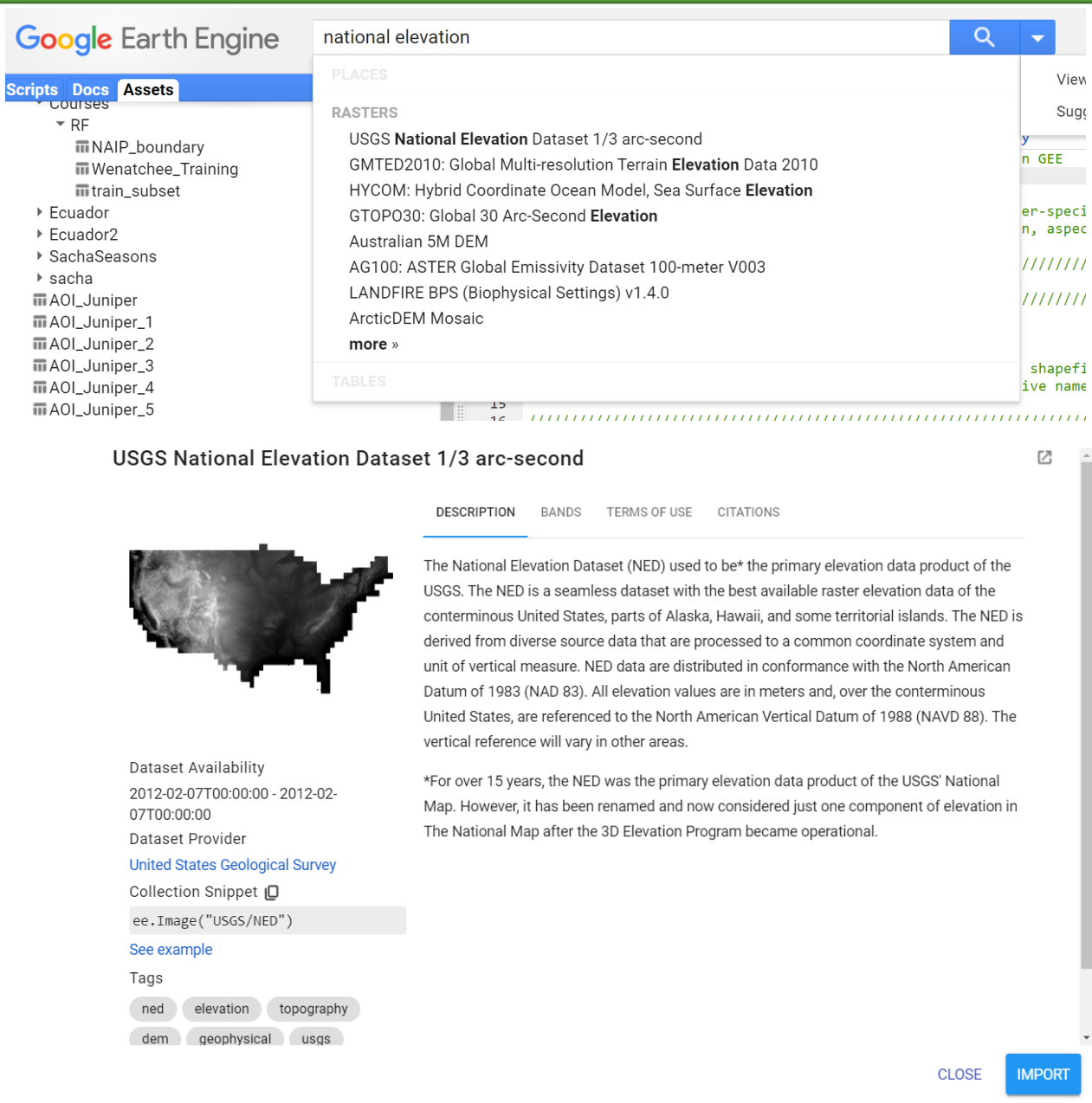
1. In the course repository, navigate to the script **ex3.1_loadTopoData** (located in DigitalSoilMapping, 02_Exercises, 01_ExerciseWorkSheets). For this exercise, we have provided an outline of a script into which you will write and copy the appropriate code. For your reference, completed scripts are provided for you as well (located in DigitalSoilMapping, 02_Exercises, 02_ExerciseCompleteScripts).
2. Open the script and inspect it. The first thing you will notice is the heading, describing the title, authorship, date modified, and summary of the script. If you scroll down, you will see headings describing the different tasks that the script plans to accomplish. But, there is no code written.
3. Save the code with a unique name to your own code repository for the course.
4. Ensure that you have imported the **VT_boundary** asset to your script and changed the name from **table** to **VT_boundary**.

B. Inspect the script

1. Take yourself on a tour through the comments in the script—get a sense of what the script plans to do. As you write your own scripts, you might choose to use comments similarly to give yourself an outline!
2. Note that on Line22 there is a comment that says "Function to add USGS 1/3 arc second topography and derive slope, aspect, and hillshade." Don't worry about this for now! Our first step, in Part C, will be to write code that accesses an elevation dataset, calculates the topographic derivatives that we're interested in. Our second step, in Part D, will be to practice turning this script into a function so that we can do the same operations on other areas without rewriting the code every time.

C. Load input DEM

1. First, we will import the digital elevation model (DEM). For the DEM, we're using a the USGS National Elevation Dataset (NED) at 1/3 arc-second spatial resolution—about 10 m.
2. You can learn more about the input dataset by searching for it in the top search bar at the top of the window. This will bring up an information window about your dataset of choice. And, it provides a code snippet that you can use to access the data. In the search bar, enter **national elevation** and select the first dataset that pops up.



3. In the left panel under the map of data that pops up, click the icon next to "Collection Snippet" to copy the code snippet to your clipboard. From here, you can paste the snippet into your script below the comment "Import NED elevation data."
4. But, don't forget to give your dataset a name! Use the **var** declaration to name the dataset in your script. You can also copy this line of code.

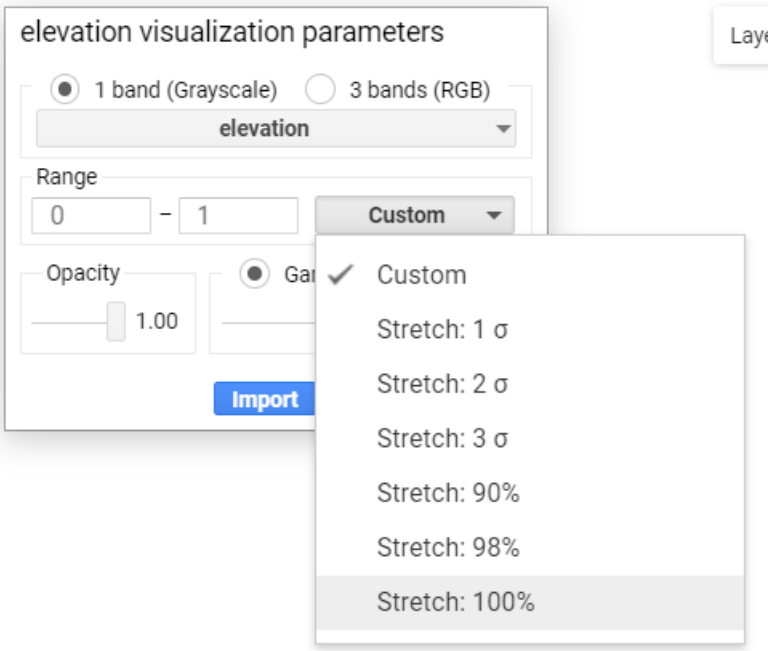
```
// Import NED elevation data

var elevation = ee.Image('USGS/NED');
```

5. So we've imported the data—but the data are for the whole continental US!
   i. (You can check this by adding this layer to the map by adding the line of code **Map.addLayer(elevation, {}, 'elevation');** and running the script. It will appear in the

layers tab. Since we haven't specified any visualization parameters, the default minimum and maximum values are 0-1 meters. The ideal range will depend on the area we visualize! You can adjust the minimum and maximum values displayed by clicking on the gear icon next to the layer name.

ii. Click the button that says "Custom." In the pane that appears, scroll down and click "Stretch: 100%". Then click Apply. The elevation layer will re-draw itself with the update parameters.



iii. Try zooming in and out and around the US, repeating the above step to adjust the parameters.

6. Since we're only interested in a small area, we'll go ahead and use the **clip** command to subset the DEM to our area of interest. Copy this line below the appropriate comment.

```
var elevation = elevation.clip(compositeArea);
```

7. You can re-run the above command to verify that our DEM was clipped to our same small study area. You may have to zoom into Vermont and adjust the visualization parameters by clicking on the gear icon next to **elevation** in the Layers drop down menu to see the clipped elevation data in the map window.

*Note: You can use the same approach at any point to inspect the layers you've just created. We've already reviewed how to **print** layers to the console; try using **Map.addLayer** to add them to the map, and then manually adjusting the visualization parameters.*

D. Calculate slope and aspect

1. Now, we need to obtain slope and aspect from our input elevation dataset. GEE has a built in function to calculate slope and aspect for an elevation layer.

2. On the left-hand side of the window, open the **Docs** pane. Navigate to **ee.Algorithms**, tab it open, and scroll down to **ee.Algorithms.Terrain**. Click to open the documentation for the function.
3. Review that this function "calculates slope, aspect, and a simple hillshade from a terrain DEM." The documentation here also notes the names and format of the output bands, and lists the arguments, or inputs, to the function. Close the window.
4. Use this function to calculate slope and aspect. Copy this line of code below the appropriate comment.

```
var topo = ee.Algorithms.Terrain(elevation);
```

5. Inspect the output of the function by printing "topo" to the console. Copy this line of code below the appropriate comment.

```
print (topo, "Topo layers");
```

6. Click **Save** and **Run**. Inspect the object in the console—tab open the object, and inspect the band names. See that the object returns bands named "elevation," "slope," "aspect," and "hillshade." We will use these bands to calculate more sophisticated topographic derivatives to use in our model.
7. You can comment out the print command by adding two backslashes at the beginning of the line, as below.

```
// Inspect topo layers
//print (topo, "Topo layers");
```

E. Calculate topo derivatives

The slope and aspect are returned in degrees, from 0-90 (refer to the function documentation for the units, and the bands attributes for the range. To make this more usable, intelligible, and standardized, we'll convert this to a percent, by first converting to radians. Here, we're using mathematical and trigonometric functions that already exist in GEE. You can see that they also store the value of pi (**Math.PI**).

Here, we're calculating more variables than we need! But, adding all of these gives us the option to pick and choose the layers that we want for our classification and regression—as well as for visualizing the layers, or for future projects.

1. Copy this code below the appropriate comment.

```
var slopeDeg = topo.select('slope');
var slopeRads = slopeDeg.multiply(Math.PI).divide(ee.Number(180));
var slopeTan = slopeRads.tan();
var slopePCT = slopeTan.multiply(ee.Number(100)).rename('slopePCT');
```

2. If you want to inspect any of these layers further, you can **print** them to the console.
3. Since aspect is also returned in degrees, we'll convert it to radians as well. We'll also calculate 8-direction aspect, as well as aspect sine and cosine from our aspect layer. 8-direction aspect

converts aspect from a continuous variable, into one where each value represents one of the cardinal directions (N, NE, E, SE, S, SW, W, NW).

4. Copy this code below the appropriate comment.

```
var aspect = topo.select('aspect');
var aspectRad = aspect.multiply(Math.PI).divide(180);
var aspectSin = aspectRad.sin().rename('sin');
var aspectCos = aspectRad.cos().rename('cos');
var aspect_8 =
(aspect.multiply(8).divide(360)).add(1).floor().uint8().rename('aspect_8');
```

5. For our last set of topo derivatives, we'll calculate three hillshades, illuminating the landscape from different directions. You won't use these as inputs for your model, but they can be useful for visualizing the landscape. Copy this code below the appropriate comment.

```
var hill_1 = ee.Terrain.hillshade(elevation,30).rename('hill_1');
var hill_2 = ee.Terrain.hillshade(elevation,150).rename('hill_2');
var hill_3 = ee.Terrain.hillshade(elevation,270).rename('hill_3');
```

6. Lastly, we'll compile the topographic derivatives into one object, using the same methods that we've used before to **select** layers and **addBands**. Copy this line of code below the appropriate comment.

```
topo = topo.select('elevation')
           .addBands([slopePCT,
                      aspectSin,
                      aspectCos]);
```

Note: we're only going to use a select few of the bands that we've calculated. You can add the other bands to this object for your own work, or future applications.

F. Add to map

1. Now, let's add all the layers to the map.

```
Map.addLayer(topo.select("elevation"), {min: 200, max: 1200}, "Elevation");
Map.addLayer(topo.select("slopePCT"), {min: 0, max: 100}, "% Slope");
Map.addLayer(topo.select("sin"), {min: -1, max: 1}, "Aspect: sin");
Map.addLayer(topo.select("cos"), {min: -1, max: 1}, "Aspect: cos");
```

2. Click **Save** and **Run**. Zoom to our study area, and inspect the different layers. Now that we've prepared all these, we can easily use them in digital soil mapping.

G. Make it a function

So, we've written all this code. But by now, having acquired the Landsat image, and calculated spectral indices, as well as topographic indices, it's starting to pile up! Luckily, there's a way to condense and save the code we've written as a function to use on other study areas—so that we don't have to write it over and over.

You've already used functions that the GEE folks and your instructors and FS colleagues have written; now you're going to learn how to write your own.

The basic idea for a function is to nest the code you've already written inside a container. You give the container a name, you tell it what inputs it takes, you tell it what to output. And then, you can access that container, or run the function.

1. Scroll up to the top of the script, under the comment that we ignored earlier in Line 22. Below it, we'll add a line that will a) declare our function, b) set input arguments, and c) begin the code that constitutes the function. Copy this line of code below the comment in Line 22.

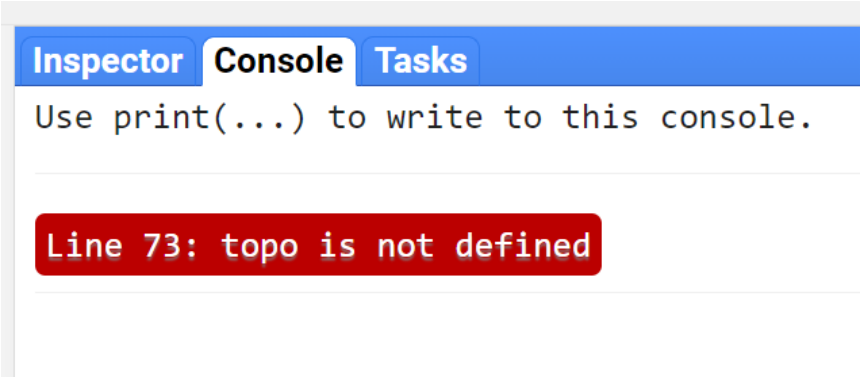
```
function getNEDTopography(compositeArea){
```

2. You'll see a red X pop up next to the line of the code—that's because we need to close the bracket.

3. Scroll down to the bottom of the code, after we Compile the selected topography bands and before we add layers to the map. Add a closing bracket **}** in a new line – for reference, this should be somewhere around line 65. It will come before the comment denoting section 3.

```
}
```

4. Click **Save** and **Run**. You will get an error—because we've written the function, but we haven't called it yet. Because the code is contained within the function, the code doesn't run unless we use the function.



5. Let's try again. After the closed bracket, add this line of code. This creates another object called "topo", by calling the function we just created.

```
var topo = getNEDTopography(compositeArea);
```

6. Oops, another error! If you didn't comment out your print command from earlier, that should still show up in the console. But, it looks like the topo variable we just created is "undefined." The problem here is that we ran the function, but we didn't tell it to give us anything out. We need to add one more line of code to get our function to work. Before your closing bracket from earlier, add this line:

```
return topo;
```

7. The bottom of your function should look like this:

```
59  // Compile selected topography bands
60  topo = topo.select('elevation')
61              .addBands([slopePCT,
62                         aspectSin,
63                         aspectCos]);
64
65  return topo;
66  }
67  var topo = getNEDTopography(compositeArea);
```

8. Click **Save** and **Run**. The layers should now populate for your area of interest—and you wrote a function!
9. The last thing we can do is to indent the whole function—to make it easier to read, and clear that the function is its own self-contained bit of code. To do this, select all the lines inside the brackets, and hit tab. All the lines should indent by the same amount, preserving the tabs you've already used in the script.

*A note about naming objects—here, we're using the same name for our **topo** object both inside the function, and outside the function. This is to make it so we don't have to adjust the commands for the layers we're adding at the end. But, these are not the same object—the function creates a topo object inside itself. And then once it runs, we must create a different object to hold the results of the function. We could call the results of the function anything (you could write the code " var bananas = get NEDTopography(compositeArea) and it would still work. In practice, it's better to have the results inside and outside the function be named different things, to mitigate confusion.*

Part 2: Calculate climate variables

A. Review the goal

1. Our next step is to calculate climate variables for our area of interest. For our last set of calculations, we practiced writing a function. For this final set of calculations, we will practice reading a complex function.
2. Let's review the goal of this script. We want to calculate meaningful climate variables for our area of interest. In this case, we are interested in calculating seasonal metrics of temperature and precipitation, over the same time period that we chose for our Landsat image composite.
3. Let's also review what we have access to. In keeping with the breadth of the GEE data catalog, we have easy access to raster data representing gridded daily temperature and precipitation observations going back to 1981, courtesy of the PRISM climate dataset.
4. So, what we need to do—is to create a script that filters and summarizes the massive dataset we have access to, into a few layers that represent climate data that will be used as predictor layers in our model.

B. Open and inspect the script

1. Open the worksheet script for exercise 3.2, located in the ExerciseWorksheets folder at DigitalSoilMapping/02_Exercises/01_ExerciseWorksheets/ex3.2_loadClimateData.

i. You can also view the completed script at DigitalSoilMapping/02_Exercises/02_ExerciseCompleteScripts/ex3.2_loadClimateData

ii. This exercise will be a little less "worksheet-y"—scroll down to see that much of the script is already written. For the parts we don't write out, we will focus on understanding.

2. Copy and save the script to the DigitalSoilMapping folder you created for the course.
3. Scroll through the complete script and orient yourself to the different sections and operations present.

i. In section 1, we again make sure that our input area is specified. Here, we also ensure that we specify the time period over which we created our Landsat composite, so that we can obtain climate data for the same time period.

ii. In section 2, we perform our climate calculations.

(a) You'll notice that there are many more indents in this script than in previous ones—this is because we have nested several conditional loops and functions in order to process data by band, by season, and by year.

(b) For example, lines 57-90 use an "**if**" loop to evaluate one of the inputs to the function that begins on line 46, and apply an appropriate calculation.

iii. A reminder that you can add in the print() command at any point to see what an output or an object looks like!

C. Specify input parameters

1. Ensure that the VT_boundary shapefile is imported to the script, using the path from your assets folder.

```
// Import shapefile from assets folder
var VT_boundary =
ee.FeatureCollection('users/YOUR_USERNAME/DigitalSoilMapping/VT_boundary');
```

2. Then, specify a composite are and composite time period.

i. The composite time period will be the same as the time period over which we calculated our image composite. Here, the composite time period gives us the ability to summarize climate data over multiple years.

```
// Specify composite area
var compositeArea = VT_boundary.geometry().bounds(); // ROI shapefile/asset or
polygon


// Specify composite time period - same range will be used for seasonal data
var year = 2019; // Start year for composite
var compositingPeriod = 0; // Number of years into the future to include
```

D. Import climate data

1. Our next step is to import the climate data we'll be using.

i. We are using the PRISM Daily Spatial Climate dataset, which provide daily values for temperature, precipitation, and vapor pressure deficit—dating back to 1981. However, these data are at fairly coarse spatial resolution (5km!). The PRISM climate data are widely used and commonly applied for continental-scale and forecasting applications. You may choose to research and import use finer-scale climate data for your map or for other mapping applications.

2. Copy this code to import the PRISM data to the script.

```
// Import PRISM daily climate data, ~5km resolution

var clim = ee.ImageCollection("OREGONSTATE/PRISM/AN81d");
```

3. Print the PRISM data to the console using the following code snippet, copied below the line where you import the code.

i. Because the PRISM dataset is so dense temporally, this specifies that we only want the first image in the dataset. This allows use to get a sense of the data structure and see what it looks like in our console.

```
print(clim.first());
```

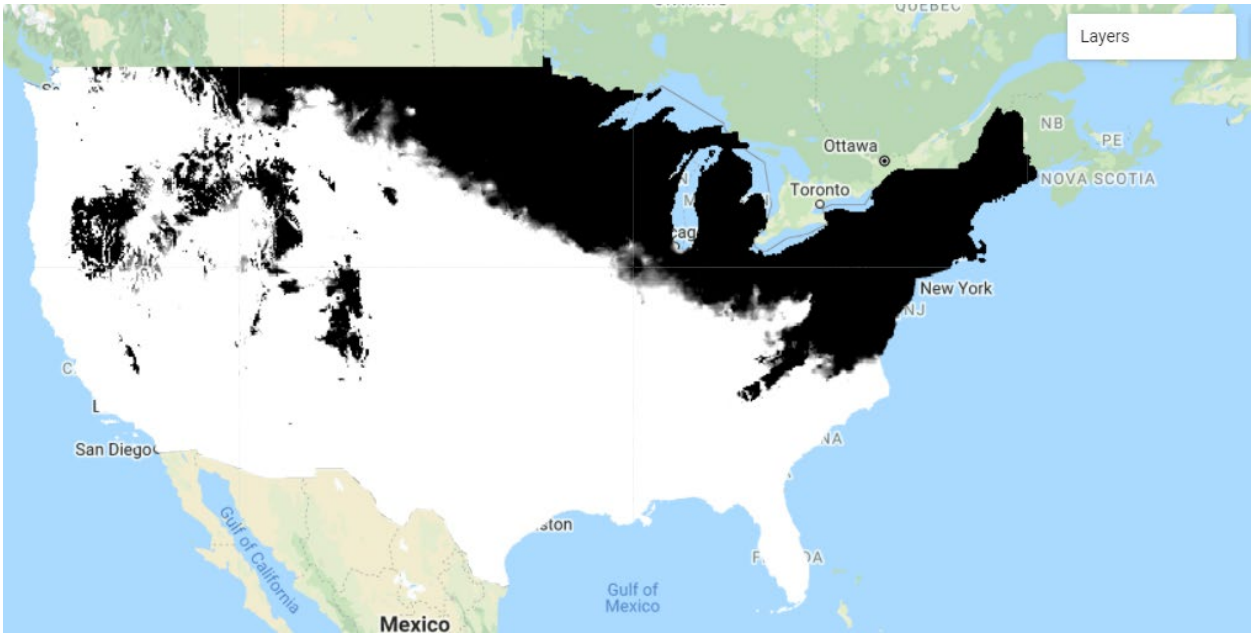
4. Inspect the band names and the properties.

i. Note that the system:index: field says 19810101. So, we selected just the first image in the PRISM dataset—the one from January 1, 1981.

5. Inspect the climate data visually by adding the layer to the map.

i. Copy this code snippet below your previous print command. Here, we're selecting the first image in the dataset, and only the tmean band. So, we're looking at the daily mean temperature for January 1, 1981.

```
Map.addLayer(clim.first().select("tmean"));
```



E. Specify function input values

1. Next, we specify the seasonal date ranges that we're interested in. Because we have daily climate data, for which the seasons are not explicitly attached, we have to specify the date ranges that indicate our seasons of interest. In addition, because our code works by stepping through each day at a time (more on that later), we also specify the length of the season.
   i. We specify these values before we begin writing our function.
   ii. The comments on season length are provided in the code for reference and clarity.

```
// Specify seasonal date ranges.
// Seasonal date ranges are based on following dates:
//winter: 12/21 - 3/19
  // - Winter is about 89 days long.
//spring: 3/20 - 6/19
  // -- Spring is 93 days.
//summer:6/20 - 9/21
  // -- Summer is 94 days.
//fall: 9/22 - 12/2
  // -- Autumn is 90 days.
```

```
var summer_startMonth = 6; //Calendar month 1-12

var summer_startDay = 20; //Day of month

var summer_length = 94; //Number of days of the season

var winter_startMonth = 12;

var winter_startDay = 21;

var winter_length = 89;
```

F. Interpret a function

Having specified our climate input and the seasons that we want to look at, we write a function to summarize our climate data by season. This part is already complete for you :) Follow along with the annotations on the screenshot below. The annotations are based on the line numbers in the screenshot—your line numbers will vary.

```
39  // Function inputs are band, seasonal dates, season length
40  function season_climate(band, month_start, day_start, season_length){
41
42    // Specify year range for calculating multi-year average
43    // Same as year range used for image composite
44    var start_year = year;
45    var end_year = year + compositingPeriod;
46    var years = ee.List.sequence(start_year, end_year);
47
48    // Calculate seasonal climate values for each year
49    return ee.ImageCollection(years.map(function(yr){
50
51      ////////////////////////////////////////////////////////////////////////
52      // For seasonal temp ('tmean' band)
53      // calculates MEAN of mean daily temperature per season, per year
54      ////////////////////////////////////////////////////////////////////////
55      if(band == 'tmean'){
56        var bandName = clim.select('tmean');
57        // filter to dates of interest
58        return bandName.filter(ee.Filter.date(
59          ee.Date.fromYMD(yr,month_start,day_start),
60          // Adds season length to starting date to get ending date
61          ee.Date.fromYMD(yr,month_start,day_start).advance(season_length,'day')))
62        // calculate mean of values over the date range
63        .mean()
64        // add time band
65        .set('system:time_start', ee.Date.fromYMD(yr, month_start,1).millis());
66      }
67
68      ////////////////////////////////////////////////////////////////////////
69      // For seasonal precip ('ppt' band)
70      // calculates SUM of total daily precip precipitation per season, per year
71      ////////////////////////////////////////////////////////////////////////
72      if(band == 'ppt'){
73        var bandName = clim.select('ppt');
74        // filter to dates of interest
75        return bandName.filter(ee.Filter.date(
76          ee.Date.fromYMD(yr,month_start,day_start),
77          // Adds season length to starting date to get ending date
78          ee.Date.fromYMD(yr,month_start,day_start).advance(season_length,'day')))
79        // calculate sum of values over the date range
80        .sum()
81        // add time band
82        .set('system:time_start', ee.Date.fromYMD(yr, month_start,1).millis());
83      }
84
85    })); // end iterating function over years
86  }
```

1. Locate the four main components of the function:
    i. Specify years for which we want to obtain climate data (Lines 45-47)
    ii. Initiate a nested function that tells the function return the climate data for each year (Line 49)
    iii. Use a conditional statement to evaluate if we have are summarizing mean temperature (Line 55), and if that's true, gather all of the temperature values for all of the days in the date range, and calculate the mean (Line 56-66).
    iv. Use a conditional statement to evaluate if we have are summarizing precipitation (Line 72), and if that's true, gather all of the precipitation values for all of the days in the date range, and calculate the sum (Line 73-83).
2. Note the closed brackets on Line 85 and Line 86.
    i. On Line 85, the closed bracket and parentheses indicates the end of the function iterating over years—the same, nested function we initiated on Line 49.
    ii. One Line 86, the closed bracket indicates the end of the **season_climate()** function we started writing on Line 40.
3. The output of this function is an image collection with one image for each year, with each image representing either the seasonal mean of temperature, or the seasonal sum of precipitation. So, for example, if we had a composite period of five years, the output image collection would have a mean seasonal temperature image for each of those five years.

G. Apply the function

Having written this function, we then have to apply it in order to get the outputs.

1. When we call the function, we again specify the following input parameters that we set out at the beginning of the function:
    i. **band**: climate variable that we're interested in
    ii. **month_start**: numerical value of the month in which to start the seasonal calculation,
    iii. **day_start**: numerical value of the day of the month in which to start the seasonal calculation,
    iv. **season_length**: number of days representing the length of the season.
2. Aside from the input parameters, the function also takes as an input the year, and number of years, we specified earlier in the script. The output of the function is an ImageCollection, where one image represents the seasonal values of our climate variable for that year.
    i. So, importantly, in the same line as we call the function, we also calculate the **mean()** across all of the images in the image collection.
    ii. As a result, an example end output is a single band that contains a multi-year mean of total winter precipitation.
    iii. (Read that a few times in case it's unclear! Temporal compositing can be confusing!)
3. Copy the code below to calculate our summarized climate variables. We apply the function four times in total: for temperature and precipitation, for both summer and winter.

```
// Calculate multi-year mean of total seasonal precipitation: summer, winter,
seasonal difference
```

```
var winter_precip = season_climate('ppt', winter_startMonth, winter_startDay,
winter_length)
                                  .mean()
                                  .rename("Winter Precip");
var summer_precip = season_climate('ppt', summer_startMonth, summer_startDay,
summer_length)
                                  .mean()
                                  .rename("Summer Precip");


// Calculate multi-year mean of mean seasonal temperature: summer, winter,
seasonal difference
var winter_tmean = season_climate('tmean', winter_startMonth, winter_startDay,
winter_length)
                                  .mean()
                                  .rename("Winter Temp");
var summer_tmean = season_climate('tmean', summer_startMonth, summer_startDay,
summer_length)
                                  .mean()
                                  .rename("Summer Temp");
```

4. We also can easily calculate seasonal differences in climate, which are bioclimatic variables that are important in determining the types of vegetation present on a landscape. This is another application of raster math, where we simply subtract winter values from summer values.

```
// Calculate seasonal climate differences
var difference_precip = summer_precip.subtract(winter_precip)
                                     .rename("Seasonal Precip Difference");
var difference_tmean = summer_tmean.subtract(winter_tmean)
                                    .rename("Seasonal Temp Difference");
```

5. Lastly, we stack the climate bands together, and clip to the boundaries of our composite area.

```
// Stack climate bands together
var climateBands = winter_precip.addBands([summer_precip,
                                           difference_precip,
                                           winter_tmean,
                                           summer_tmean,
```

```
                                        difference_tmean]);


// Clip to ROI boundaries
climateBands = climateBands.clip(compositeArea);
```

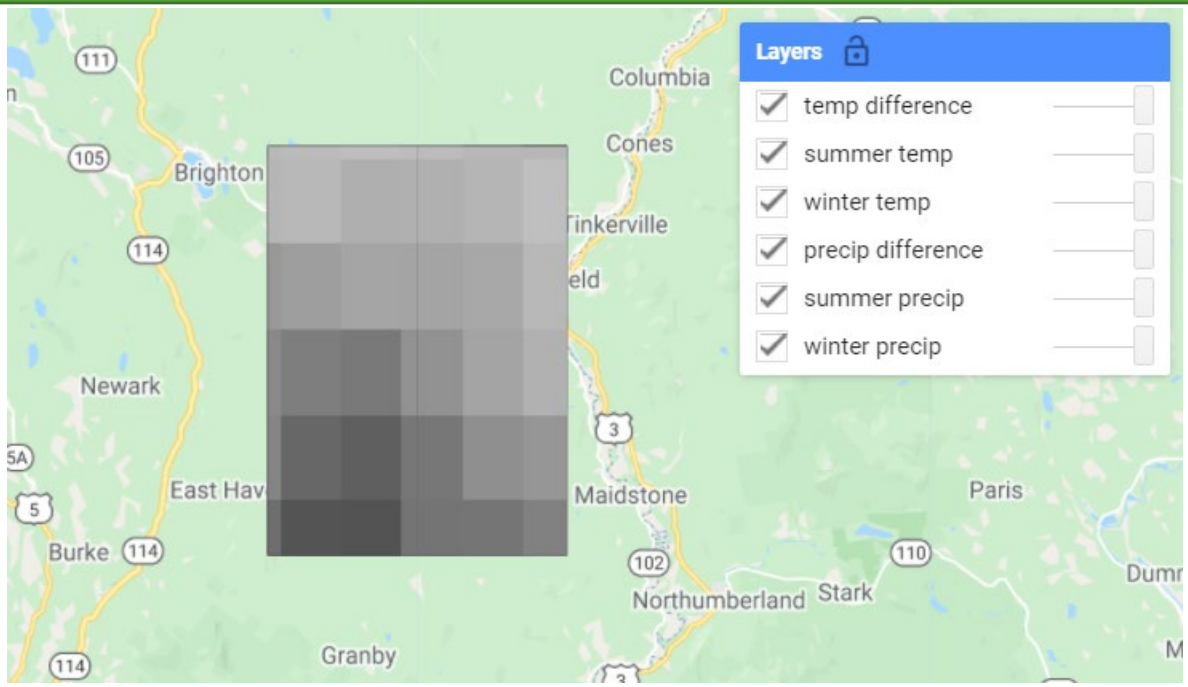
H. Add to the map

1. And finally, we add the layers to the map and visualize.

```
// Visualize precip layers
Map.addLayer(climateBands.select('Winter Precip'), {min: 130, max: 450}, 'winter
precip');
Map.addLayer(climateBands.select('Summer Precip'), {min: 130, max: 450}, 'summer
precip');
Map.addLayer(climateBands.select('Seasonal Precip Difference'), {min: 50, max:
150}, 'precip difference');
// Visualize temp layers
Map.addLayer(climateBands.select('Winter Temp'), {min: -10, max: 0}, 'winter
temp');
Map.addLayer(climateBands.select('Summer Temp'), {min: 15, max: 20}, 'summer
temp');
Map.addLayer(climateBands.select('Seasonal Temp Difference'), {min: 20, max: 25},
'temp difference');
```

I. Review and run the script

1. Go back through the script and make sure you understand what each line is doing. Feel free to add your own comments to enhance your understanding!
2. Click Save and Run. Inspect the results in the map viewer. Again, remember that this is at 5km resolution.
    i. To see what this looks like at the continental scale, you can comment out the command to clip the data to the ROI boundaries. Click Save and Run. You'll need to adjust the visualization parameters, like we did with the DEM, to view these appropriately over larger spatial extents.



Congratulations! You've calculated all of the input predictor layers for your soil classification and regression.