A matter of function

samuel gachuhi

2023-11-18

# Introduction

In programming, [functions](https://users.cs.utah.edu/~germain/PPS/Topics/functions.html) help in automation of processes. A function generally takes in data, processes it, and returns a result. Functions are really useful when you have a repetitive task of the same procedure with expected results. If let’s say, you wanted to analysed 20 satellite images. For each of these images, you want to clip it to a certain country, extract images within a certain date range, and select those below a certain cloudy pixel percentage threshold. Functions would not only help you save time (and maybe sanity), but would reduce human errors that comes with fatigue of repetitive programming. Furthermore, since the end goal is an image below a certain cloud coverage, within a certain date range and country, this is already a replicable framework fit for a function!

In today’s blog, we shall look at how to create Google Earth Engine (GEE) functions. These are just simple functions, but in reality, those for GEE can be incredibly complex. Think of one that does scaling, masking and index calculations and you get the idea.

# GEE Workflow

## Loading the data

Obviously, there has to be some data to start with. In our case, we are going to use a shapefile comprising Kenya’s county administrative regions. Yours truly has been generous to provide it [here](https://code.earthengine.google.com/?asset=users/sammigachuhi-test/kenya_counties). Clicking on the link should direct you on how to make the shapefile available in your asset’s tab.

From the assets tab, import the shapefile into your script and rename it to counties. Our first step? Display Kenya’s administrative regions on our GEE canvas.

//////////////////////////////////////////////////////////////////////////////////////////////////////////  
// Load Kajiado county into GEE  
///////////////////////////////////////////////////////////////////////////////////////////////////////////  
Map.addLayer(counties, {}, "Counties of Kenya");

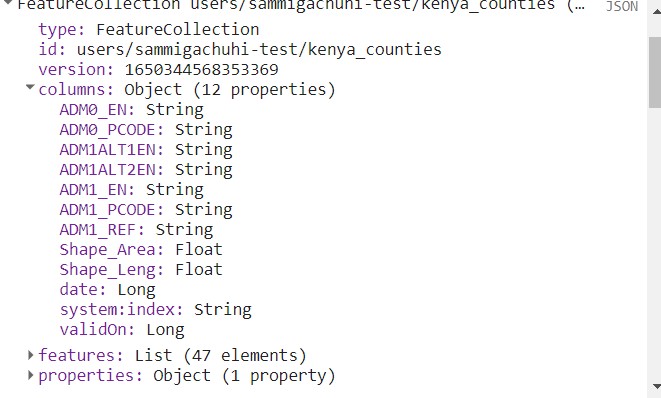
It’s also good to have an inkling of some of their names. Forty-seven counties is no mean memory task. I know all of them!

/// Print columns of counties data  
print("Counties of Kenya: ", counties);

What we have just done is print out the attributes of our counties shapefile. On your console, you should get the below print out:

knitr::include\_graphics("E:/documents/gis800\_articles/GEE\_project/functions\_pics/county\_attributes.jpg")

## Warning in  
## knitr::include\_graphics("E:/documents/gis800\_articles/GEE\_project/functions\_pics/county\_attributes.jpg"):  
## It is highly recommended to use relative paths for images. You had absolute  
## paths:  
## "E:/documents/gis800\_articles/GEE\_project/functions\_pics/county\_attributes.jpg"



## Building intuiton

There are two ways to create functions. The first step involves breaking down the workflow into small bits. Once you confirm that the workflow works perfectly, you can add up every step into a function. This is the step that I generally follow. The second involves creating a function straight from your head, but I usually recommend it if you are really sure of what you’re doing, coupled by much experience.

Going by the first suggestion, we want to select Kajiado county from the administrative shapefile, print out the attributes of this county, and display the county on the GEE canvas. These are the steps we would follow.

// Select Kajiado county only from `counties` feature collection  
var kajiadoCounty = ee.FeatureCollection("users/sammigachuhi-test/kenya\_counties")  
 .filter(ee.Filter.eq("ADM1\_EN", "Kajiado"));  
   
print("Metadata for Kajiado county: ", kajiadoCounty);  
  
// Zoom to Kajiado county  
Map.centerObject(kajiadoCounty, 8);  
  
// Add Kajiado county to the map  
var kajiadoVisParams = {  
 strokeWidth: 6  
};  
Map.addLayer(kajiadoCounty, kajiadoVisParams, "Kajiado County");

Now imagine you wanted to automate the above process, right from loading the entire country shapefile to selecting a particular county, and displaying both its confines and metadata. How would you go about it? Simple! Just combine the above steps into the function keyword.

## Create a function for vector data

Starting from scratch, you would assign a name next to the function keyword. This would be the function name.

// Create a function that loads a county data, and selects a county in one line!  
function getCounty(parameter) {  
 // do something  
 return output;  
}

Our function is called getCounty. The name is descriptive of what we want. It could be any name, getASoda, getGoing, getSmart etc. Just give a simple descriptive name that hints on the purpose. The parameter represents the values that will get into the function. You can think of parameters as what will enable the function perform its work. Parameters in a function are separated by brackets.

Having had a short intro to functions, what do you think will be our parameters to enable us select a county or any sub-administrative region, say from a shapefile containing hundreds of such regions? Here’s a clue: you would need the variable of the country shapefile, the name of the county (or sub-administrative region of interest), and the name of the attribute field which contains the names of the administrative regions of interest. You see, with these parameters, our function can work for any country, or even continent!

// Create a function that loads a county data, and selects a county in one line!  
function getCounty(countyDataset, countyName, attributeName) {  
 }

Next, we can pass the commands that will enable us generate our output. We wanted a specific county from the large country set. The same procedures we followed when doing it line by line are what will be added inside the parentheses {}.

Breaking it into chunks, we will start with the very first action: loading the country dataset.

function getCounty(countyDataset, countyName, attributeName) {  
 // Add the county dataset to GEE  
 Map.addLayer(countyDataset, {}, countyDataset + " Feature Layer");

A semi-colon : indicates an end of statement. But this does not mean we are done with our function yet. We are just informing the reader our first statement is already over. It could be one line, two lines or more but in our case our first step of loading the dataset was a one liner.

To print out the metadata of our county shapefile, we proceed as follows:

function getCounty(countyDataset, countyName, attributeName) {  
 // Add the county dataset to GEE  
 Map.addLayer(countyDataset, {}, countyDataset + " Feature Layer");  
 // Print metadata of county dataset feature collection  
 print("Metadata of " + countyDataset + " dataset: ", countyDataset);

Ensure you maintain the indentation. Additionally, when creating functions, it helps in writing comments on what each statement was doing –both for your future self and clueless reader.

For the rest of the steps, such as selecting the county, printing out its metadata and displaying it respectively, we add the following steps.

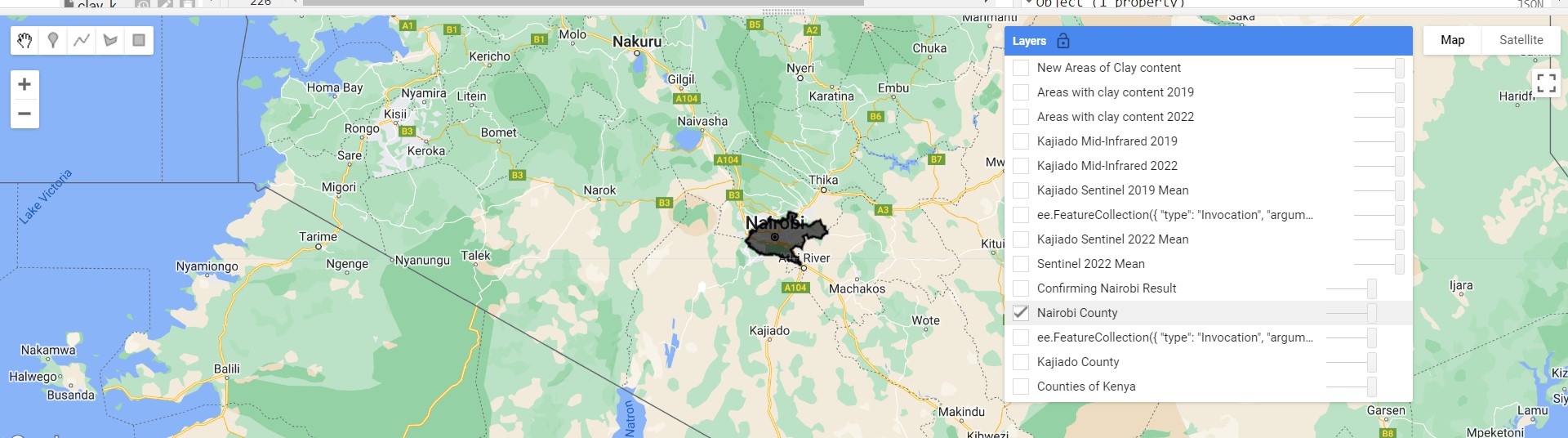
function getCounty(countyDataset, countyName, attributeName) {  
 // Add the county dataset to GEE  
 Map.addLayer(countyDataset, {}, countyDataset + " Feature Layer");  
 // Print metadata of county dataset feature collection  
 print("Metadata of " + countyDataset + " dataset: ", countyDataset);  
 // Select the specified county name  
 var selectedCounty = countyDataset.filter(ee.Filter.eq(attributeName, countyName));  
 // Print metadata of selected county  
 print("Metadata for " + countyName + " county: ", selectedCounty);  
 // Zoom to the selected county  
 Map.centerObject(selectedCounty, 8);  
 // Add selected county to the map  
 var countyVisParams = {  
 strokeWidth: 6  
 };  
 Map.addLayer(selectedCounty, countyVisParams, countyName + " County");  
 // Return the selected county for further analysis if you wish  
 return selectedCounty;  
   
}

Now, you can pass this function to any country dataset to get a particular sub-administrative region. We first started with Kajiado county, but now, I changed my mind to view Nairobi county. The function will do everything in a whiz. We will assign it to a variable called nairobi where we can do with it as we please, but for now, let’s call it a day here.

var nairobi = getCounty(counties, "Nairobi", "ADM1\_EN");  
Map.addLayer(nairobi, {}, "Confirming Nairobi Result");  
print("Just confirming Nairobi metadata: ", nairobi);

knitr::include\_graphics("E:/documents/gis800\_articles/GEE\_project/functions\_pics/nairobi\_gee\_functions.jpg")

## Warning in  
## knitr::include\_graphics("E:/documents/gis800\_articles/GEE\_project/functions\_pics/nairobi\_gee\_functions.jpg"):  
## It is highly recommended to use relative paths for images. You had absolute  
## paths:  
## "E:/documents/gis800\_articles/GEE\_project/functions\_pics/nairobi\_gee\_functions.jpg"



The only problem is that both in the console and GEE canvas the selected country dataset is referred to as ee.FeatureCollection({blah...blah...blah}). But that is not our major problem for now. Learning functions is the first thing to beat.

## Create a function to handle raster data

Compared to vector data, raster data is harder to deal with. This is because when dealing with raster, some processing is required prior to analysing it. For this section, we want to create a function that filters the image collection to select images falling within the bounds of our feature collection, within a particular date range and below cloudy coverage percentage. Furthermore, we will calculate the mean pixel values for each band within our target date range, clip it to the boundaries of our feature collection and display the image. As advised, you do each step individually. The last step will only be appending each of the steps followed.

We start by selecting images that intersect a given feature, our shapefile in this case, filter to within a particular date range, and select those below a certain cloudy pixel percentage.

////////////////////////////////////////////////////////////////////////////////////  
// Get clear Sentinel-2 images from Kajiado county  
////////////////////////////////////////////////////////////////////////////////  
var kajiadoSentinel = ee.ImageCollection("COPERNICUS/S2\_SR\_HARMONIZED")  
 // Filter the collection to the bounds of Kajiado county  
 .filterBounds(kajiadoCounty)  
 // Select images for the year 2022  
 .filterDate("2022-01-01", "2022-03-31")  
 // Filter to dry months of Kajiado - January - March  
 .filter(ee.Filter.calendarRange(1, 3, "month"))  
 // Select images with CLOUDY PIXEL PERCENTAGE less than 5  
 .filter(ee.Filter.lte("CLOUDY\_PIXEL\_PERCENTAGE", 5));

Let’s do some data exploration of what we’ve got so far. Getting the metadata of each image and the size of the images that have passed the litmus test is some modest exercise.

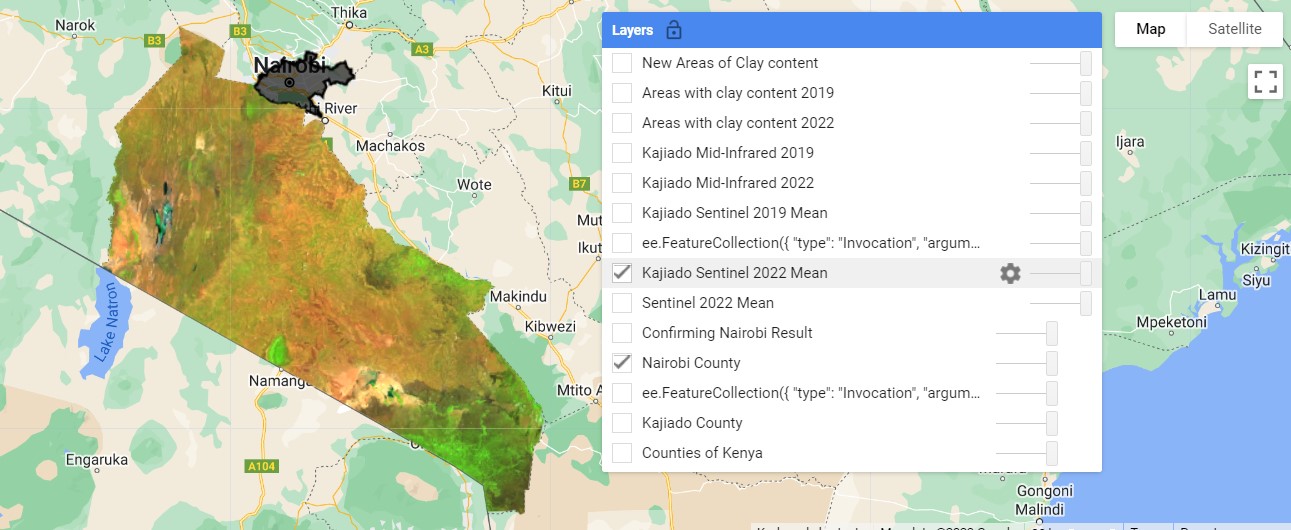
// Print out the Sentinel-2 images over Kajiado county  
print("Sentinel-2 images with cloud cover less than 5% in Kajiado county: ", kajiadoSentinel);  
  
// Get the number of images clipped to Kajiado county  
print("No. of Sentinel-2 images with cloud cover <5% over Kajiado county: ", kajiadoSentinel.size());

For calculating the mean values and clipping the images to the county boundaries, the below steps seal the deal.

// Get the mean value of every band over Kajiado county in 2022  
var mean2022 = kajiadoSentinel.mean();  
  
// Add the mean of 2022  
var mean2022Vis = {  
 bands: ['B11', 'B8', 'B3'],  
 min: 0,  
 max: 4000  
};  
  
Map.addLayer(mean2022, mean2022Vis, "Sentinel 2022 Mean");  
  
// Clip the image to the boundaries of Kajiado county  
var kajiadoMean2022 = mean2022.clipToCollection(kajiadoCounty);  
  
// Add the clipped image to the canvas  
Map.addLayer(kajiadoMean2022, mean2022Vis, "Kajiado Sentinel 2022 Mean");

knitr::include\_graphics("E:/documents/gis800\_articles/GEE\_project/functions\_pics/kajiado\_mean\_2022.jpg")

## Warning in  
## knitr::include\_graphics("E:/documents/gis800\_articles/GEE\_project/functions\_pics/kajiado\_mean\_2022.jpg"):  
## It is highly recommended to use relative paths for images. You had absolute  
## paths:  
## "E:/documents/gis800\_articles/GEE\_project/functions\_pics/kajiado\_mean\_2022.jpg"



Now to create a function that performs the above steps in one go, we follow the same exact procedure as we did for vectors. Of course a function name has to go first, but we can also introduce new variables inside a function, especially when one variable is used an input in another process.

function filterSentinelToFeature(selectedCounty,   
 startYear, endYear, startMonth, endMonth, pixelPercentage) {  
 // The image collecion of Sentinel-2 images  
 var images = ee.ImageCollection("COPERNICUS/S2\_SR\_HARMONIZED")  
 // Filter the collection to the bounds of the selected county  
 .filterBounds(selectedCounty)  
 // Select images between the start and end year in Google Earth Engine string format  
 .filterDate(startYear, endYear)  
 // Filter to the start and end months (in numerical form)  
 .filter(ee.Filter.calendarRange(startMonth, endMonth, "month"))  
 // Select Sentinel-2 images with CLOUDY PIXEL PERCENTAGE less than 5  
 .filter(ee.Filter.lte("CLOUDY\_PIXEL\_PERCENTAGE", pixelPercentage));  
 // Print out metadata of all images meeting the above conditions  
 print("Metadata of all S2 images for years (" + startYear + "-" + endYear + ") between months " +   
 startMonth + " and " + endMonth + " with cloud pixel percentage < " + pixelPercentage + "%",   
 images);  
 // Get the mean of all the bands from the images within the selected date ranges  
 var meanImages = images.mean();  
 // Clip the mean of all the bands to the boundaries of the selected county  
 var meanImagesClipped = meanImages.clipToCollection(selectedCounty);  
 // Add the visualization parameters  
 var meanVisParams = {bands: ['B11', 'B8', 'B3'], min: 0, max: 4000};  
 // Add the clipped image containing the mean of all bands to the canvas  
 Map.addLayer(meanImagesClipped, meanVisParams, selectedCounty + " Sentinel 2022 Mean");  
 // Return the clipped image containing the mean of all the bands  
 return meanImagesClipped;  
   
}

In the last three expressions, we introduce three new variables, namely: meanImages, meanImagesClipped and meanVisParams. We can introduce new variables inside a function to avoid duplication of tasks, and to prevent confusion. We also use variables especially when one, most or all of them will be used inside another function within the function. Yes, functions can hold other functions within them but that’s a story for another day.

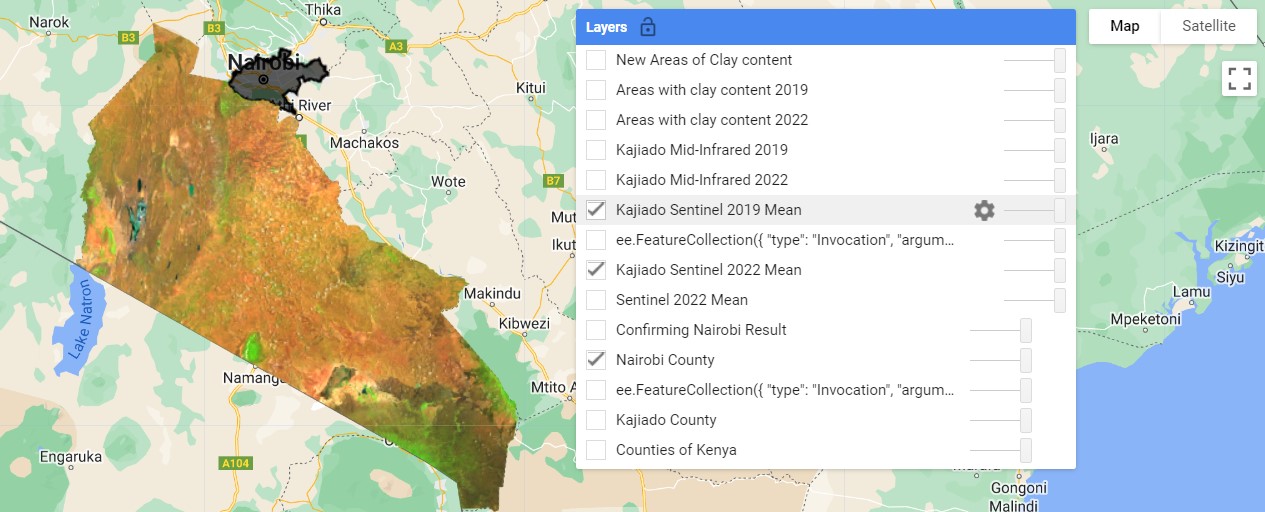
We have also introduced one small but crucial component in almost all JavaScript functions: the return statement. The return statement stop the execution of a function and *returns* a value. My teacher’s used to tell us never describe a word with the same word. Think of return as giving you what it has received. In keeping with our dialogue, once the resulting data has been displayed on GEE, you get the clipped image as the result. This result can be saved into a variable for further analysis.

To prove the credibility of our function, we decided to pass it 2019 images. It did so without breaking a sweat.

var kajiadoMean2019 = filterSentinelToFeature(kajiadoCounty, "2019-01-01", "2019-03-31", 1, 3, 5);  
Map.addLayer(kajiadoMean2019, mean2022Vis, "Kajiado Sentinel 2019 Mean");  
  
// Print the metadata for each mean image  
print("Metadata for mean of 2022: ", kajiadoMean2022);  
print("Metadata for mean of 2019: ", kajiadoMean2019);

knitr::include\_graphics("E:/documents/gis800\_articles/GEE\_project/functions\_pics/kajiado\_mean\_2019.jpg")

## Warning in  
## knitr::include\_graphics("E:/documents/gis800\_articles/GEE\_project/functions\_pics/kajiado\_mean\_2019.jpg"):  
## It is highly recommended to use relative paths for images. You had absolute  
## paths:  
## "E:/documents/gis800\_articles/GEE\_project/functions\_pics/kajiado\_mean\_2019.jpg"



We have reached the end of creating functions in GEE. Learning to create functions was part of a much grander plan of calculating the clay content in my otherwise semi-arid county. The formula for achieving the results was borrowed from this [article].

The full code used in creating functions as well as calculating the clay content in each of the two years is available [here](https://code.earthengine.google.com/f89e3daea28064a90e78b1e12decd7b0). I would particularly appreciate help in the last section where I attempted to proof the difference of clay content, which actually increased from 2019 to 2022 from the clayEmergence variable which was achieved by subtracting the 2019 mask from the 2022 mask. Particularly the crap coding from line 218 downwards.

For further teaching on functionalization in GEE, this [NEON tutorial](https://www.neonscience.org/resources/learning-hub/tutorials/intro-gee-functions) nails most concepts home.

# Conclusion

In this somewhat short tutorial, we discovered the purposes of functions and when best to use them. We saw how to create a function that deals with vector images, and went a step further to create a function that deals with rasters. Dealing with images added more complexity, such as the need of filtering to below a certain cloudy pixel percentage. Finally, we passed each function to a variable, namely nariobi and kajiadoMean2019 respectively.