

## GEOG 676 - GIS Programming Lab 6 Instructions

### **Pre-lab reading:**

#### **Map creation:**

#### **Renderers**

Renderers are used by ArcGIS to determine how feature data gets drawn on the screen. Data is just data until we have a renderer that tells the program exactly how that data should be presented to the user. The four basic feature renderer types are SimpleRenderer, UniqueValueRenderer, GraduatedColorsRenderer, and GraduatedSymbolsRenderer. Below we will dive into what each renderer is and present a use case for why you may want to use a specific renderer.

It is important to note that not all layers support renderers, so always make sure to check that the layer has a renderer attribute before trying its renderer.

#### **SimpleRenderer**

The most basic of renderers, the SimpleRenderer is used to draw all features of a layer the same. This is the default renderer used whenever we add a feature class to a map; you know the plain, all-one-color features look. We will not bother diving into how to use this renderer as they are the default and they tend to be rather boring.

#### **UniqueValueRenderer**

UniqueValueRenderer is used to distinguish features in a feature layer based off of some classification, usually a data field. With this renderer, we are given programmatic control over the type of field used in the classification of features as well as the color ramp used when drawing said features.

Below you will see a code example of a UniqueValueRenderer in use.

## GraduatedColorsRenderer

GraduatedColorsRenderer is used to render features in a feature layer based off of a specific field in the feature class. We can use this particular renderer for creating choropleth maps programmatically; useful if you need to quickly produce such maps. This renderer gives us control over the style of choropleth produced; allowing the user to determine how many bins are present as well as the color ramp used.

Below you will see a code example of a GraduatedColorsRenderer in use.

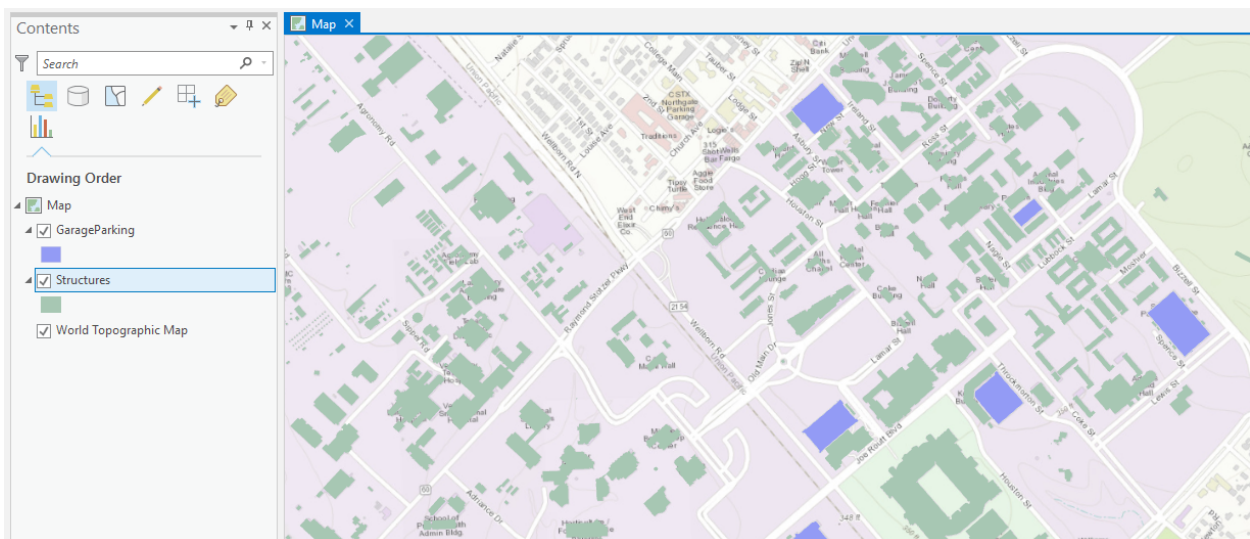
## GraduatedSymbolsRenderer

The GraduatedSymbolsRenderer is quite similar to GraduatedColorsRenderer in that each let you render features in a feature class depending on a number of bins and let you specify the color ramp used. This style of renderer is useful for creating graduated symbol styled maps.

## Making a map with UniqueValueRenderer

Our example for UniqueValueRenderer will focus on automating the creation of a unique value map using the Structures feature class present in the Campus.gdb from previous lectures.

The first thing you will want to do is open up ArcGIS Pro and add a folder connection to the geodatabase that contains our feature class. Once the connection is made, we need to add the Structures feature class to the map. Save the project .aprx to a location you can reference using arcpy. Your map should look something like this:



With the .aprx saved, we can start on our Python script. The first step of any script is to import the arcpy module. With the module imported, we need to get a reference to our .aprx project file. Using the built-in arcpy method ArcGISProject(), we simply provide the path to the file and we have our reference. After we reference the .aprx, we create a variable that holds a reference to the first map in our project. To do so, we use the listMaps() function to get our campus map shown above. The parameter used inside the function listMaps() is the name given by default to a new map inside a newly created project.

```
import arcpy

project = arcpy.mp.ArcGISProject(r"C:/tmp/ArcGISPython/" + r"\Mod23.aprx")

campus = project.listMaps('Map')[0]
```

Now that we have a reference to the map, we can loop through the layers present in said map. Using the method listLayers() we can get a list of all layers in the map from which we can iterate through. The first thing we want to do is make sure the layer is a feature layer. We can check using a simple conditional that checks the current layer's isFeatureLayer attribute: if it is true we continue to set the renderer. The next step is to create a copy of the current feature layer's symbology. Symbology is the attribute that contains a layer's renderer. We copy the symbology to make the naming a little easier to read.

A very important step when altering a layer's renderer in code is to make sure the layer has an attribute renderer before proceeding. If we try to reference a layer's renderer without it actually containing one, we'll end up raising an exception. The last check we need to do is to make sure the current layer's name is "Structures", if our map contains multiple layers we want to change the renderer on the correct one.

Now that we have the feature class we want, we can finally start to change our renderer. The whole process is surprisingly quick and simple if you choose to let arcpy decide the colors and some of the other finer details. With our variable symbology we simply call updateRenderer() and provide the string value "UniqueValueRenderer". This will set the renderer of symbology equal to UniqueValueRenderer. Keep in mind that this has only changed the renderer of the COPY symbology, not the actual layer's symbology. We'll address that in a moment. Once the renderer is updated, we need to let arcpy know what field we want to symbolize our features with. We can set that with symbology.renderer.fields and set the value of that equal to a list that contains the field we want to symbolize our features. Now then, set the layer's symbology back to the symbology copy we updated. Without this line we will not see any changes. The last step is to simply save a copy of our project's .aprx as a new project if we desire and we're done!



The full code with comments is found below.

```
import arcpy

# Reference to our .aprx
project = arcpy.mp.ArcGISProject(r"C:/tmp/ArcGISPython/" + r"\Mod23.aprx")

# Grab the first map in the .aprx
campus = project.listMaps('Map')[0]

# Loop through available layers in the map
for layer in campus.listLayers():

    # Check that the layer is a feature layer
    if layer.isFeatureLayer:

        # Obtain a copy of the layer's symbology
        symbology = layer.symbology

        # Makes sure symbology has an attribute "renderer"
        if hasattr(symbology, 'renderer'):

            # Check if the layer's name is "Structures"
            if layer.name == "Structures":

                # Update the copy's renderer to be "UniqueValueRenderer"
                symbology.updateRenderer('UniqueValueRenderer')

                # Tells arcpy that we want to use "Type" as our unique value
                symbology.renderer.fields = ["Type"]
```

```

        # Set the layer's actual symbology equal to the copy's
        layer.symbology = symbology # Very important step
    else:
        print("NOT Structures")

project.saveACopy(r"C:/tmp/ArcGISPython/" + r"\Mod23b.aprx")

```

## **Making a map with GraduatedColorsRenderer**

The next example we'll dive into is using a GraduatedColorsRenderer to create a choropleth map. Many of the same ideas and code used in the previous example will be used here. We start by creating a new project (.aprx) file and add in a data layer. For this map, we'll be using the GarageParking feature class and creating a choropleth based off the Shape\_Area data field. Add the layer to the map and save to the usual location. Create a new, plain Python script and import arcpy. Create a variable that references the newly created project file. With that done, make a variable that references the map that contains your data layer in the project. At this point, you should be here:

```

import arcpy

project = arcpy.mp.ArcGISProject(r"C:/tmp/ArcGISPython/" + r"\Mod23.aprx")

campus = project.listMaps('Map')[0]

```

Once more we need to perform some checks to make sure we are dealing with a feature class, it is the correct feature class, and it has a renderer attribute. First, we loop through the available layers returned by the function listLayers(). On each layer we check to make sure the current layer is a feature class; if so we copy that layer's symbology to a new variable called symbology. The last check we need to perform is to make sure the name of the layer is the layer we want to change the renderer of. If we've found the right layer we can change it's renderer. Remember that we make the changes to the symbology variable first, then set the layer's layer.symbology attribute equal to our symbology variable. Using updateRenderer() we pass in a value of GraduatedColorsRenderer. After that, we set the symbology.renderer.classificationField, an attribute not used in our UniqueValueRenderer example, equal to the data field we will be using to determine our classification. Now with our classification field set, we need to tell arcpy how many bins we wish to create; for this example we are going to use five bins. Our renderer is almost set! Now we need to set a color ramp, else our choropleth will look quite dull. We can query our project for a list of available color ramps by using the listColorRamps() function without any parameters. We'll be using a simple orange color ramp. Since listColorRamps() returns a list of values, we will use the square brackets to tell it we want the first value in the list of returned color ramps. The last renderer setup step is to set the layer.symbology equal to our symbology variable and we should be golden! Remember to save your resulting project.



```
import arcpy

# Reference to our .aprx
project = arcpy.mp.ArcGISProject(r"C:/tmp/ArcGISPython/" + r"\Mod23.aprx")

# Grab the first map in the .aprx
campus = project.listMaps('Map')[0]

# Loop through available layers in the map
for layer in campus.listLayers():

    # Check if layer is a feature layer
    if layer.isFeatureLayer:

        # Obtain a copy of the layer's symbology
        symbology = layer.symbology

        # Check if it has a 'renderer' attribute
        if hasattr(symbology, 'renderer'):

            # Check if the layer's name is 'GarageParking'
            if layer.name == "GarageParking":

                # Update the copy's renderer to be 'GraduatedColorsRenderer'
                symbology.updateRenderer('GraduatedColorsRenderer')

                # Tell arcpy which field we want to base our choropleth off
                symbology.renderer.classificationField = "Shape_Area"

                # Set how many classes we'll have
                symbology.renderer.breakCount = 5

                # Set the color ramp
```

```

        symbology.renderer.colorRamp =
project.listColorRamps('Oranges (5 Classes)')[0]

        # Set the layer's actual symbology equal to the copy's
        layer.symbology = symbology # Very important step
    else:

        print("NOT GarageParking")

project.saveACopy(r"C:/tmp/ArcGISPython/" + r"\Mod23c.aprx")

```

### **Arcpy messaging & progress:**

Back when we were covering how to create and add a tool into a toolbox we mentioned a method inside the tool called `updateMessages()`. We glanced over the subject at the time as it wasn't too important in the actual creation of the toolbox, in fact it isn't even a required method like `execute` or `__init__`. Though its a completely optional tool method, we will now be going over why you should be utilizing `updateMessages()` and how to go about using it.

Have you written a Python script yet or any program for that matter and been presented with an error message that left you scratching your head? You know those errors that say `ERROR 999999: An exception has occurred and you think to yourself "Thanks for the heads up, now what is actually wrong"; those are always fun. When you make a tool you don't want to be a part of that problem; you don't want to be the person associated with those types of errors. Below we'll dive into how to create meaningful error messages from your toolbox.`

### **The toolbox**

For this lecture we will be adding a new input to the toolbox we created previously for seeing which buildings were near a particular building. We want to be able to display a message to inform the user if they try to buffer a building that does not exist inside the campus Structures feature class.

Below you will find all the code needed for the toolbox from the previous lecture.

```

import arcpy

class Toolbox(object):
    def __init__(self):
        """Define the toolbox (the name of the toolbox is the
name of the
.pyt file)."""
        self.label = "GEOG676_Tools"
        self.alias = "GEOG676_Tools"

        # List of tool classes associated with this toolbox
        self.tools = [BuildingProximity]

class BuildingProximity(object):
    def __init__(self):
        """Define the tool (tool name is the name of the
class)."""
        self.label = "Building Proximity"
        self.description = "Determines which buildings on TAMU's
campus are near a targeted building"
        self.canRunInBackground = False # Only used in ArcMap
        self.category = "Building Tools"

    def getParameterInfo(self):
        """Define parameter definitions"""
        param0 = arcpy.Parameter(
            displayName="Building Number",
            name="buildingNumber",
            datatype="GPString",
            parameterType="Required",
            direction="Input"
        )
        param1 = arcpy.Parameter(
            displayName="Buffer radius",
            name="bufferRadius",
            datatype="GPDoube",
            parameterType="Required",
            direction="Input"
        )
        param1.filter.type = "Range"
        param1.filter.list = [10, 100]
        params = [param0, param1]
        return params

    def isLicensed(self):

```



```

        """Set whether tool is licensed to execute."""
        return True

    def updateParameters(self, parameters):
        """Modify the values and properties of parameters before
internal
        validation is performed. This method is called whenever
a parameter
        has been changed."""
        return

    def updateMessages(self, parameters):
        """Modify the messages created by internal validation
for each tool
        parameter. This method is called after internal
validation."""
        return

    def execute(self, parameters, messages):
        """The source code of the tool."""
        campus =
r"D:/DevSource/Tamu/GeoInnovation/_GISProgramming/data/modules/1
7/Campus.gdb"

        # Setup our user input variables
        buildingNumber_input = parameters[0].valueAsText
        bufferSize_input = int(parameters[1].value)

        # Generate our where_clause
        where_clause = "Bldg = '%s'" % buildingNumber_input

        # Check if building exists
        structures = campus + "/Structures"
        cursor = arcpy.SearchCursor(structures,
where_clause=where_clause)
        shouldProceed = False

        for row in cursor:
            if row.getValue("Bldg") == buildingNumber_input:
                shouldProceed = True

        # If we shouldProceed do so
        if shouldProceed:
            # Generate the name for our generated buffer layer
            buildingBuff = "/building_%s_buffed_%s" %
(buildingNumber_input, bufferSize_input)
            # Get reference to building

```

```

        buildingFeature = arcpy.Select_analysis(structures,
campus + "/building_%s" % (buildingNumber_input), where_clause)
        # Buffer the selected building
        arcpy.Buffer_analysis(buildingFeature, campus +
buildingBuff, bufferSize_input)
        # Clip the structures to our buffered feature
        arcpy.Clip_analysis(structures, campus +
buildingBuff, campus + "/clip_%s" % (buildingNumber_input))
        # Remove the feature class we just created
        arcpy.Delete_management(campus + "/building_%s" %
(buildingNumber_input))
    else:
        print("Seems we couldn't find the building you
entered")
    return None

```

### **Adding in an error message**

The function we'll be adding code to is the `updateMessages()` method inside our `BuildingProximity` class.

```

def updateMessages(self, parameters):

    """Modify the messages created by internal validation for
each tool

    parameter. This method is called after internal
validation."""

    return

```

As you can see from the method declaration, we have two arguments: `self` and `parameters`. Remember from the toolbox lecture that the `parameters` argument passed in here are the parameters we defined back in the `getParameterInfo()` method. Let us go ahead and loop through the parameters so we can find the parameter with a name of `buildingNumber`. This is the name we gave the parameter that holds our building number input.

```

def updateMessages(self, parameters):

    for param in parameters:

        if param.name == "buildingNumber":

            # We've found the correct parameter

            buildingNum = param.value

    return

```

With our parameter found, we now need to see if the value provided exists inside the Structures layer. In order to do so we need to use a SearchCursor and count how many results we get back. We need to redefine our campus and where\_clause variables so we can access the data layers. We then create a cursor variable that contains the results of the SearchCursor.

```
def updateMessages(self, parameters):
    for param in parameters:
        if param.name == "buildingNumber":
            buildingNum = param.value

            campus =
r"D:/DevSource/Tamu/GeoInnovation/_GISProgramming/data/modules/2
4/Campus.gdb"

            where_clause = "Bldg = '%s'" % buildingNum

            cursor = arcpy.SearchCursor(campus + "/Structures",
where_clause=where_clause)

            return
```

Now that we have our results, how do we tell how many rows we have in the cursor? For whatever reason, arcpy does not provide a simple way to determine how many rows are in a cursor and you cannot use len() on cursors either. The simplest way would be to iterate through a for loop and count how many times the for loop executes. It's not the most efficient or the nicest way, but it works. Then we simply check to see if the count is greater than 0; if it is, we know we have that building in the feature class Structures. Let's add all this in now.

```
def updateMessages(self, parameters):
    for param in parameters:
        if param.name == "buildingNumber":
            buildingNum = param.value

            campus =
r"D:/DevSource/Tamu/GeoInnovation/_GISProgramming/data/modules/2
4/Campus.gdb"

            where_clause = "Bldg = '%s'" % buildingNum
```

```

        cursor = arcpy.SearchCursor(campus + "/Structures",
where_clause=where_clause)

        count = 0

        for row in cursor:

            count += 1

        if count == 0:

            param.setErrorMessage("Cannot find building %s
in Structures" % buildingNum)

        return

```

We are not interested in how many rows are returned, we only care if there are no rows returned by the SearchCursor(). If no rows are found, we know that the building is not in the Structures feature class and we write out a message. To do so we use the setErrorMessage() method on the parameter. We simply give it an error message and test it out!

## Error

In addition to telling us an error has occurred, we can also use the following line to output messages to the Results.

```
arcpy.AddMessage("Buffering...")
```

When placed inside the tool method execute(), this will print out a message that looks like the following:

## Results

### **Adding a progressor**

Another element you can add to a tool to improve the overall experience is to add in a progressor. What is a progressor? A progressor is the little bar that lets you know the tool is currently working on something. There are two types of progressor: default and step. The default progressor displays a moving bar that constantly moves back and forth. This is good to use when we do not know how long something will take or if the operation takes a while.

Step is the progressor type we will create below. With step we can advance the progressor in increments after a specific piece of code has executed.

## Progressor

Adding in one of these is fairly simple as all we need to do is set it and periodically change the position and label. Let's go ahead and add a progressor into our tool.

Inside of the execute() method, go ahead and define the following variables:

```
readTime = 1.5
start = 0
maximum = 100
step = 25
```

readTime is used to delay the progressor titles by a small margin so that the user can actually read them without the text flashing away. The variable start defines the beginning position of our progressor, maximum defines the absolute maximum value, and step is used to move the progressor along.

Once our variables are defined we can setup the progressor and advance it after key portions of our tool have finished executing. Setting up a progressor involves calling `arcpy.SetProgressor()` and providing in five parameters: the type of progressor, the progressor label, the start value, the end value, and a step value.

# Setting up the progressor

```
arcpy.SetProgressor("step", "Checking building proximity...", start, maximum, step)
```

Once set up, we can then use the following method code to advance the progressor:

```
arcpy.SetProgressorPosition(start + step)
arcpy.SetProgressorLabel("Validating building number once more...")
```

```
time.sleep(readTime)
```

The method `SetProgressorPosition` changes the "percentage" completed of the progressor while `SetProgressorLabel` changes the message displayed alongside the progressor. The last line involves importing the time module; we fix this with `import time` just below `import arcpy`. When we call `time.sleep(readTime)` we are momentarily halting the execution of our tool. This is used just so we can see the progressor labels changing.

## Progressor

```
execute()
```

Below you will find the entire `execute()` method of our newly improved tool.

```
def execute(self, parameters, messages):
    """The source code of the tool."""
    # Define our progressor variables
    readTime = 2.5
    start = 0
    maximum = 100
    step = 25

    # Setup the progressor
    arcpy.SetProgressor("step", "Checking building
proximity...", start, maximum, step)
    time.sleep(readTime)
    # Add message to the results pane
    arcpy.AddMessage("Checking building proximity...")

    campus =
r"D:/DevSource/Tamu/GeoInnovation/_GISProgramming/data/modules/1
7/Campus.gdb"
```

```

# Setup our user input variables
buildingNumber_input = parameters[0].valueAsText
bufferSize_input = int(parameters[1].value)

# Generate our where_clause
where_clause = "Bldg = '%s'" % buildingNumber_input

# Check if building exists
structures = campus + "/Structures"
cursor = arcpy.SearchCursor(structures,
where_clause=where_clause)
shouldProceed = False

# Increment the progressor and change the label; add
message to the results pane
arcpy.SetProgressorPosition(start + step)
arcpy.SetProgressorLabel("Validating building number
once more...")
time.sleep(readTime)
arcpy.AddMessage("Validating building number once
more...")

for row in cursor:
    if row.getValue("Bldg") == buildingNumber_input:
        shouldProceed = True

# If we shouldProceed do so
if shouldProceed:

```

```

        # Generate the name for our generated buffer layer
        buildingBuff = "/building_%s_buffed_%s" %
(buildingNumber_input, bufferSize_input)

        # Get reference to building
        buildingFeature = arcpy.Select_analysis(structures,
campus + "/building_%s" % (buildingNumber_input), where_clause)

        # Buffer the selected building
        arcpy.Buffer_analysis(buildingFeature, campus +
buildingBuff, bufferSize_input)

        # Increment the progressor, change label, output
message to results pane too
        arcpy.SetProgressorPosition(start + step)
        arcpy.SetProgressorLabel("Buffering...")
        time.sleep(readTime)
        arcpy.AddMessage("Buffering...")

        # Clip the structures to our buffered feature
        arcpy.Clip_analysis(structures, campus +
buildingBuff, campus + "/clip_%s" % (buildingNumber_input))

        # Increment the progressor, change label, output
message to results pane too
        arcpy.SetProgressorPosition(start + step)
        arcpy.SetProgressorLabel("Clipping...")
        time.sleep(readTime)
        arcpy.AddMessage("Clipping...")

        # Remove the feature class we just created
        arcpy.Delete_management(campus + "/building_%s" %
(buildingNumber_input))

        # Increment the progressor, change label, output
message to results pane too
        arcpy.SetProgressorPosition(maximum)
        arcpy.SetProgressorLabel("Cleaning up files...")

```



```

        time.sleep(readTime)
        arcpy.AddMessage("Cleaning up files...")
    else:
        print("Seems we couldn't find the building you
entered")
    return None

```

### **Arcpy advanced tool parameters**

When we worked on our toolbox in the earlier lectures, we assumed the user would only be working from the **Structures** layer found in the Campus geodatabase. Assumptions are good for simple scripts, but once you create a tool from your script, the user is probably going to want to supply a layer for input. Or what if you've created a tool that works off of several layers? If your tool requires people to modify the underlying Python in order for them to use it, they probably won't keep using such a tool.

In order to fix such issues, we will dive into some of the other tool inputs one can use inside a tool. Unfortunately there are no one-size-fits-all solutions for parameter inputs, so it is up to the developer to select an input parameter type that best fits the required inputs.

### **Advanced input parameter types**

#### **Multivalue**

Just as the name suggests, **multiValue** allows your tool to take several values for the same input. Say you have a tool who needs to create several buffers of varying sizes, you would need a multivalue input. Or if you needed to clip several layers, you wouldn't want the user to add them in by editing your script, you'd just provide a multivalue parameter.

```

def getParameterInfo(self):
    """Define parameter definitions"""
    param0 = arcpy.Parameter(
        displayName="Input Features",
        name="in_features",
        datatype="GPFeatureLayer",
        parameterType="Required",

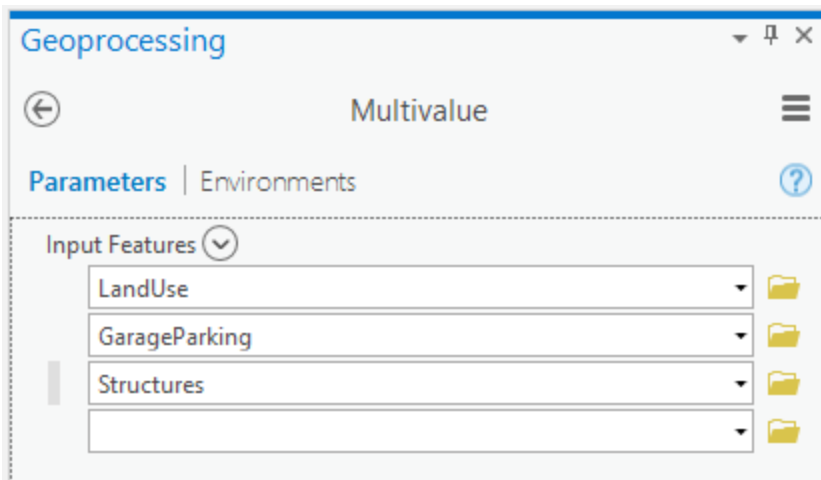
```

```

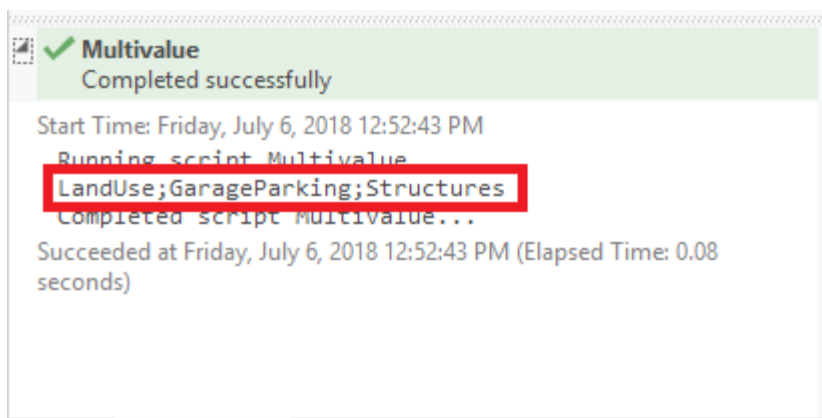
        direction="Input",
        multiValue=True)

    params = [param0]
    return params

```



When you use a multivalue parameter, getting the value from said parameter can now be a little tricky. If we get the value using **param0.valueAsText**, we will end up with a semi-colon separated string containing all the values provided to the multivalue input. It is up to you to then split the string to get the actual values.



Below you can find some code that will separate out the different values present in the multivalue parameter.

```

def execute(self, parameters, messages):
    """The source code of the tool."""
    # Iterate through all parameters

```

```

    for param in parameters:
        # Set a variable equal to the string value of the
current parameter
        paramVal = param.valueAsText

        # Split the variable paramVal on semi-colons; this gives
us a list of inputs provided to the multivalue param

        tokens = paramVal.split(";")

        # Iterate through all the inputs in the list (the
tokens)

        for token in tokens:
            # Print out each token value

            arcpy.AddMessage(token)

    return

```

## Value Table

A **Value table** is a type of input that allows you to specify multiple entries. These entries can be any of the standard parameter data types such as feature classes, fields, strings, et cetera.

```

def getParameterInfo(self):
    param0 = arcpy.Parameter(
        displayName='Input Features',
        name='in_features',
        datatype="GPFeatureLayer",
        parameterType='Required',
        direction='Input')

    param1 = arcpy.Parameter(
        displayName='Statistics Field(s)',
        name='stat_fields',
        datatype='GPValueTable',
        parameterType='Required',

```

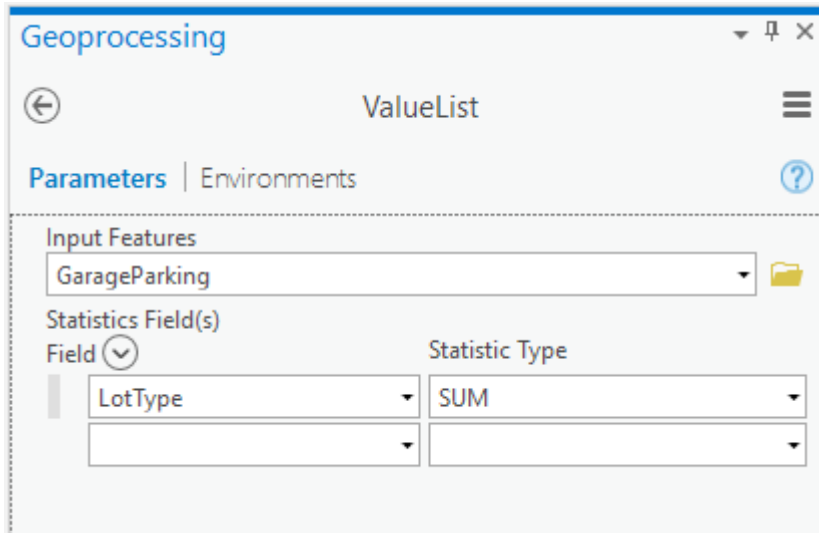
```

        direction='Input')

    param1.parameterDependencies = [param0.name]
    param1.columns = [['Field', 'Field'], ['String', 'Statistic
Type']]
    param1.filters[1].type = 'ValueList'
    param1.values = [['NAME', 'SUM']]
    param1.filters[1].list = ['SUM', 'MIN', 'MAX', 'STDEV',
'MEAN']
    params = [param0, param1]
    return params

def execute(self, parameters, messages):
    """The source code of the tool."""
    for param in parameters:
        paramText = param.valueAsText
        tokens = paramText.split(" ")
        layer = tokens[0]
        field = tokens[1]
        stat_field = tokens[2]
    return

```



You can get the values from a value table by splitting the parameter value with a single space character. This will return three strings: the layer given as input, the field chosen and the statistical field value selected.

### Composite parameters

Sometimes you will want your tool parameter to accept a variety of different data types requiring you to use what's known as a **composite parameter**. A **composite parameter** is a parameter that accepts multiple data types. These data types are defined as a list and set to the parameters **datatype** attribute.

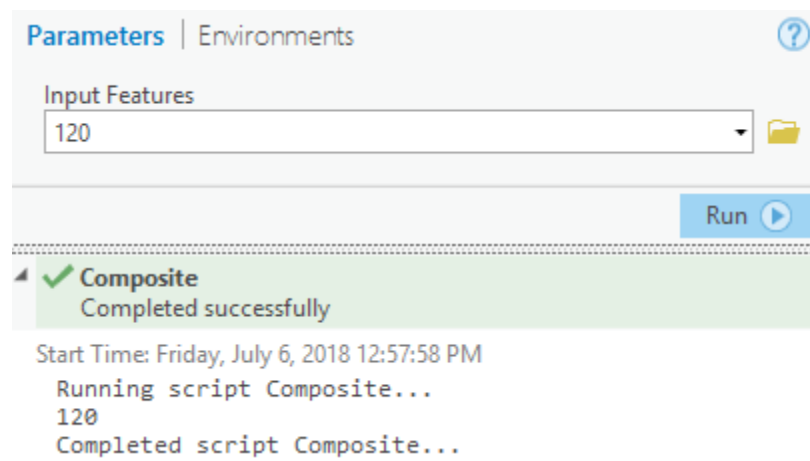
```
def getParameterInfo(self):
    """Define parameter definitions"""
    param0 = arcpy.Parameter(
        displayName="Input Features",
        name="in_features",
        datatype=["GPFeatureLayer", "GPLayer",
"GPRasterDataLayer", "GPLong"],
        parameterType="Required",
        direction="Input")
    params = [param0]
    return params
```

```

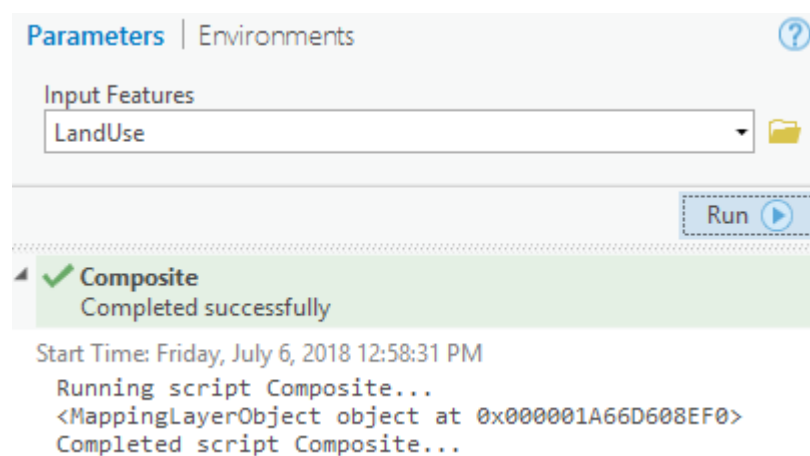
def execute(self, parameters, messages):
    """The source code of the tool."""
    for param in parameters:
        # This will print out the value for each parameter,
        regardless of its datatype
        arcpy.AddMessage(param.valueAsText)
    return

```

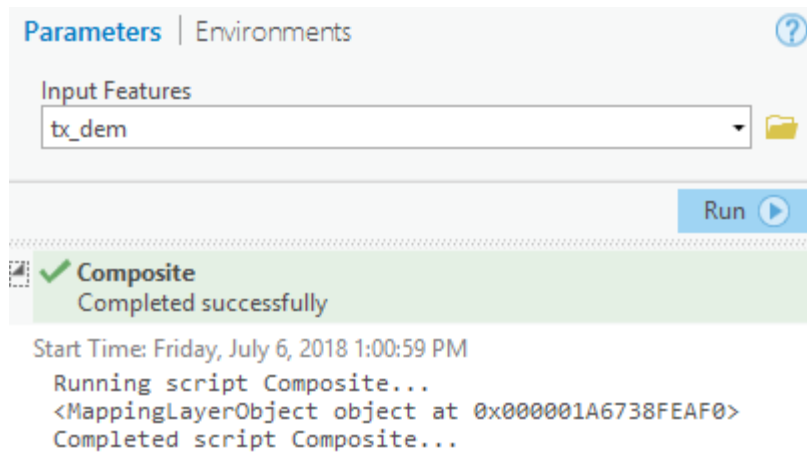
A composite parameter such as the one defined above handles a variety of different input types without issue. The three images below show how this single parameter handles all these input types without breaking a sweat.



An integer as an input to the parameter



A feature class as an input to the parameter



A raster layer as an input to the parameter

## Derived

You can also have a parameter that is known as **derived**. These parameters depend on a previous parameter being fulfilled before they *activate*. Consider the following example:

```
def getParameterInfo(self):  
    """Define parameter definitions"""  
    param0 = arcpy.Parameter(  
        displayName="Input Features",  
        name="in_features",  
        datatype="GPFeatureLayer",  
        parameterType="Required",  
        direction="Input")  
  
    param1 = arcpy.Parameter(  
        displayName="Output Features",  
        name="out_features",  
        datatype="GPFeatureLayer",  
        parameterType="Derived",  
        direction="Output")
```

```

param1.parameterDependencies = [param0.name]

params = [param0, param1]
return params

def execute(self, parameters, messages):
    """The source code of the tool."""
    for param in parameters:
        arcpy.AddMessage(param.valueAsText)
    return

```

The parameter called **param1** has a parameterType of **Derived**. This means that **param1** is dependent on some other parameter. The next line uses the **parameterDependencies** attribute to set which parameter is param1 dependent on; in this case param0.

Getting out the value of a derived parameter is the same as getting a value out of any plain parameter type; we just use the **value** or **valueAsText** attributes.

## Default values

In addition to having the user input the value of a parameter, we can also set default values for the convenience of the user based off their environment settings. To set a default value, simply set the parameters **value** attribute within the **getParameterInfo()** method. You may see usage of a particular parameter attribute **defaultEnvironmentName**. This particular attribute is used for setting the default environment for the parameter, not a default value.

```

def getParameterInfo(self):
    param0 = arcpy.Parameter(
        displayName="Input Workspace",
        name="in_workspace",
        datatype="DEWorkspace",
        parameterType="Required",
        direction="Input")

```



```

param1 = arcpy.Parameter(
    displayName="Buffer radius",
    name="bufferRadius",
    datatype="GPLong",
    parameterType="Required",
    direction="Input")

# In the tool's dialog box, the first parameter will show
# the workspace environment's value (if set)
param0.defaultEnvironmentName = "workspace"

# The default value for the buffer will be 25.5. This will
show every time the tool is run.

param1.value = 25.5
params = [param0, param1]
return params

def execute(self, parameters, messages):
    """The source code of the tool."""
    for param in parameters:
        arcpy.AddMessage(param.valueAsText)

    return

```

Just because a default value has been set does not mean that the user cannot change the value whenever they run the tool; defaults are supposed to make tool usage a little easier and less tedious.

## **Advanced filters**

### **Value List**

The **Value List** filter is useful for providing a set of values that the user can then choose from. A **Value List** differs from a multiValue in that with a multiValue you are free to select as many options as you want. A **Value List** limits your user's options to whatever you have specified ahead of time.

```
def getParameterInfo(self):
    param0 = arcpy.Parameter(
        displayName="Input Features",
        name="in_features",
        datatype="GPLong",
        parameterType="Required",
        direction="Input")

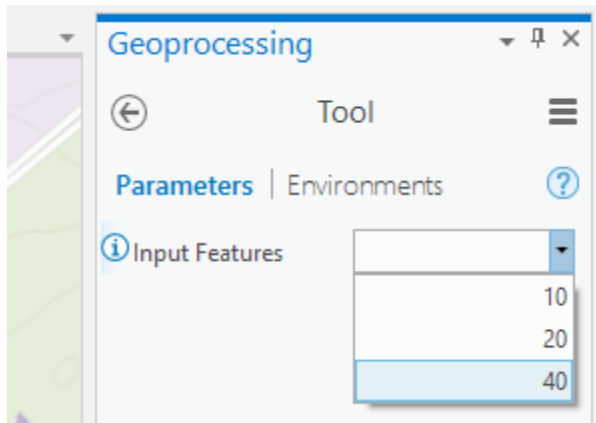
    param0.filter.type = "ValueList"
    param0.filter.list = [10, 20, 40]

    params = [param0]
    return params

def execute(self, parameters, messages):
    """The source code of the tool."""
    for param in parameters:
        arcpy.AddMessage(param.valueAsText)

    return
```

To get the value out of a value list style parameter, we access the **value** or **valueAsText** attribute of the parameter.



## Feature Class

We can also use filters to limit the type of feature the user provides in the form of an input layer. Say we want to perform a clip somewhere in our tool code. What if a user passes in a point feature class? We could have code that checks for such situations, but we can also use a filter that will do the work for us.

```
def getParameterInfo(self):
    param0 = arcpy.Parameter(
        displayName="Input Features",
        name="in_features",
        datatype="GPFeatureLayer",
        parameterType="Required",
        direction="Input")
    param0.filter.list = ["Polygon"]
    params = [param0]
    return params

def execute(self, parameters, messages):
    """The source code of the tool."""
    for param in parameters:
        arcpy.AddMessage(param.valueAsText)

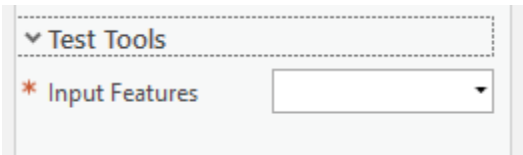
    return
```

We simply set the **filter.list** attribute of **param0** equal to a list that contains those geometry types that will work with the tool; in this case polygons. Since we are only limiting the type of geometry we are accepting, we can still access the parameter's value with **value** or **valueAsText** attributes.

## Categories

If you have a large amount of input's, you can categorize your inputs into collapsible sections. In order to do so, you only need to set the **category** attribute on your parameter to a string value. Those parameters with the same value for **category** will appear in the same section.

```
def getParameterInfo(self):  
    """Define parameter definitions"""  
    param0 = arcpy.Parameter(  
        displayName="Input Features",  
        name="in_features",  
        datatype="GPLong",  
        parameterType="Required",  
        direction="Input")  
  
    param0.filter.type = "ValueList"  
    param0.filter.list = [10, 20, 40]  
    param0.category = "Test Tools"  
    params = [param0]  
    return params  
  
def execute(self, parameters, messages):  
    """The source code of the tool."""  
    for param in parameters:  
        arcpy.AddMessage(param.valueAsText)  
  
    return
```



Categories do not alter parameters in any way, so we can use **value** or **valueAsText** to access the properties value.






### Lab06 Tasks:

- Create a script that can generate either a unique value or graduated color map
- Turn said script into a toolbox that can be accessed from the Geoprocessing pane in ArcGIS Pro
- Utilize a progressor inside the tool to inform the user how far along the script is in generating the map

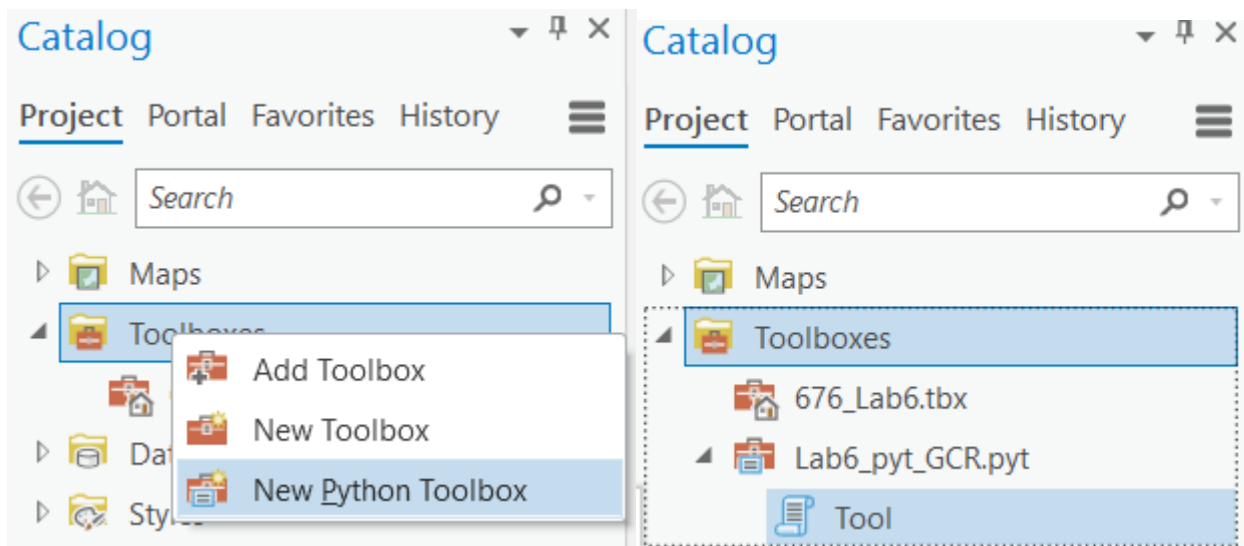
For this homework assignment, you will need to create a toolbox whose job is to create either a GraduatedColorsRenderer or UniqueValueRenderer based map. Once you have a working toolbox, you will need to add in a progressor who's label tracks what portion of the code is executing. This means the progressor should increment every so often and the label text should change with each progression.

### STEP 1: Generate a map

1. Create a new ArcGIS Pro project.
2. Add Campus.gdb and load one feature (GarageParking layer)
3. Save the project. You'll notice a toolbox (tbx) file within the project folder and in the Catalog tab.

	ArcPro_project.gdb	10/21/2018 4:29 PM	File folder	
	ImportLog	10/21/2018 3:48 PM	File folder	
	Index	10/21/2018 4:29 PM	File folder	
	ArcPro_project.aprx	10/21/2018 3:48 PM	ArcGIS Project File	21 KB
	ArcPro_project.tbx	10/21/2018 3:45 PM	ArcGIS Toolbox	4 KB

4. To make an editable toolbox, right click Toolboxes and select New Python Toolbox. Name it after the tool you choose (GraduatedColorsRenderer or UniqueValueRenderer) and you should see it added to your project with a generic tool inside the dropdown. We're going to change the toolbox to make this tool serve a unique purpose!



## STEP 2: Edit your toolbox

1. At this point I recommend making a copy of your toolbox and saving it as a .py so that you can edit it more easily (with colors). Don't forget to change it back to a .tbx when you're done.
2. Do not edit anything besides the tool name in the first part, otherwise the toolbox will not work.

```
import arcpy

class Toolbox(object):
    def __init__(self):
        """Define the toolbox (the name of the toolbox is the name of the .pyt file)."""
        self.label = "Toolbox"
        self.alias = ""

        # List of tool class associated with this toolbox
        self.tools = [GarageSize]

class GarageSize(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label = "Garage Area"
        self.description = "Classifies the given garages by area. Larger area classes are designated by darker colors"
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define parameter definitions"""
```

## Your tool's name

3. Now add your parameters. At minimum you'll need:
  - a. Input project
  - b. Layer to classify (which layer you want to use to generate a color map)
  - c. Output location
  - d. Output project name

The first one should look something like this:

```
class GarageSize(object):
    def __init__(self):
        """Define the tool (tool name is the name of the class)."""
        self.label= "Garage Area"
        self.description = "Classifies the given garages by area. Larger area classes are designated by darker colors"
        self.canRunInBackground = False

    def getParameterInfo(self):
        """Define paramater definitions"""

        param0 = arcpy.Parameter(                                     #Original ArcGIS Pro Project Name
            displayName="Input ArcGIS Pro Project Name",
            name="aprxInputName",
            datatype="DEFile",
            parameterType="Required",
            direction="Input"
        )
```

Check the link at the bottom of the instructions for your data types.

4. Next is the progressor within the execute section. a. First, you'll define progressor variables

```
def execute(self, parameters, messages):
    """The source code of the tool."""

    #Define Progressor Variables
    readTime = 2.5
    start = 0
    max = 100
    step = 33
```

- b. Set up your progressor with a message of your choice and read in your project file for your input project parameter.

```

#Setup Progressor
arcpy.SetProgressor("step", "Validating Project File...", start, max, step)
time.sleep(readTime)
#Add message to the results pane
arcpy.AddMessage("Validating Project File...")

project = arcpy.mp.ArcGISProject(parameters[0].valueAsText)

#Grabs the first instance of a map from the .aprx
campus = project.listMaps('Map')[0] #user navigates to this specified folder

```

c. Now that the progressor is initialized, set up new labels as it increments through the tool. You'll need to do this again after your for loop classification.

```

# Increment Progressor
arcpy.SetProgressorPosition(start + step)
arcpy.SetProgressorLabel("Finding your map layer...")
time.sleep(readTime)
arcpy.AddMessage("Finding your map layer...")

```

## 5. For loop

a. To classify your layer, you'll need to set up the following filters:

```

#loop through the layers of the map
for layer in campus.listLayers():
    #Check if the layer is a feature layer
    if layer.isFeatureLayer:
        #Copy the layer's symbology
        symbology = layer.symbology
        #Make sure symbology has renderer attribute
        if hasattr(symbology, 'renderer'):
            #Check layer name
            if layer.name == parameters[1].valueAsText: #user will have to input this as an exact string

                #Increment Progressor
                arcpy.SetProgressorPosition(start + 2*step)
                arcpy.SetProgressorLabel ("Calculating and classifying...")
                time.sleep(readTime)
                arcpy.AddMessage("Calculating and classifying...")

                #update the copy's renderer to "Graduated Colors Renderer"
                symbology.updateRenderer('GraduatedColorsRenderer')
                #Tell arcpy which field we want to base our choropleth off of
                symbology.renderer.classificationField = "Shape_Area"
                #Set how many classes we'll have
                symbology.renderer.breakCount = 5
                #set color ramp
                symbology.renderer.colorRamp = project.listColorRamps('Greens (5 Classes)')[0]
                #Set the layer's actual symbology equal to the copy's
                layer.symbology = symbology
            else:
                print("NOT Structures")

```



- b. Make sure to add your own labels and decide what field you'll use for classification.
- c. You can also adjust the break count and color to your personal preference!

6. Increment your progressor again to end the process. Save a copy of your classification using your last parameters.

```
project.saveACopy(parameters[2].valueAsText + "\\" + parameters[3].valueAsText + ".aprx") #param 2 is the folder location and param 3 is the name of the new project
return
```

#### **STEP 4: Submission**

- Screenshot of your .py code with Terminal window illustrating that it can be run without any errors.
- Screenshot of your Toolbox in ArcGIS Pro with no error messages popping up after running.
- Link to your Github page that contains the Python codes (Python script and toolbox)

#### **Notes:**

- If you're having difficulty with one (GraduatedColorsRenderer or UniqueValueRenderer) try the other one before giving up.
- Make sure you're checking if your toolbox is saved as a .py or a .pyt
- Keep your workspace clean. If you make copies of the toolbox to try different code, make sure you stay organized and remember where you save everything.
- Triple check that your data type is correctly listed in your parameters! \*\*\*\*
- [https://pro.arcgis.com/en/pro-app/arcpy/geoprocessing\\_and\\_python/defining-parameter-data-types-in-a-python-toolbox.htm](https://pro.arcgis.com/en/pro-app/arcpy/geoprocessing_and_python/defining-parameter-data-types-in-a-python-toolbox.htm)